SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Optimism
Prepared by: Sherlock
Lead Security Expert: obront, Trust
Dates Audited: January 23 - February 6, 2023
Prepared on: March 3, 2023

Introduction

Anyone can say it, only Optimism can back it up. Bedrock is the culmination of countless hours of research and development with the goal of building the best rollup architecture yet.

Scope

The key components of the system can be found in our monorepo at commit 3f4b3c3281.

- L1 Contracts
- L2 Contracts (AKA Predeploys)
- op-node
- op-geth (in its own repo)

Findings

Each issue has an assigned severity:

- Specification error findings are restricted to factually incorrect information only, i.e. statements which describe something different from what the implementation actually does.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Specification	Medium	High
25	13	3

Issues not fixed or acknowledged

Specification	Medium	High
0	0	0

Security experts who found valid issues

lemonmon
obront
cmichel
shw
ck
Bahurum
csanuragjain
Robert
keccak123
Ro
tnch
Koolex
usmannk
unforgiven
0xdeadbeef
supernova
0x1337
rvierdiiev
tsvetanovv
ustas
Bobface
Allarious
Barichek
GalloDaSballo
HE1M
Bnke0x0
xiaoming90
ladboy233
cergyk

Issue H-1: Malicious user can finalize other's withdrawal with less than specified gas limit, leading to loss of funds

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/109

Found by

obront

Summary

Transactions to execute a withdrawal from the Optimism Portal can be sent with 5122 less gas than specified by the user, because the check is performed a few operations prior to the call. Because there are no replays on this contract, the result is that a separate malicious user can call `finalizeWithdrawalTransaction()` with a precise amount of gas, cause the withdrawer's withdrawal to fail, and permanently lock their funds.

Vulnerability Detail

Withdrawals can be initiated directly from the `L2ToL1MessagePasser` contract on L2. These withdrawals can be withdrawn directly from the `OptimismPortal` on L1. This path is intended to be used only by users who know what they are doing, presumably to save the gas of going through the additional more "user-friendly" contracts.

One of the quirks of the `OptimismPortal` is that there is no replaying of transactions. If a transaction fails, it will simply fail, and all ETH associated with it will remain in the `OptimismPortal` contract. Users have been warned of this and understand the risks, so Optimism takes no responsibility for user error.

However, there is an issue in the implementation of `OptimismPortal` that a withdrawal transaction can be executed with 5122 gas less than the user specified. In many cases, this could cause their transaction to revert, without any user error involved. Optimism is aware of the importance of this property being correct when they write in the comments:

> We want to maintain the property that the amount of gas supplied to the call to the target contract is at least the gas limit specified by the user. We can do this by enforcing that, at this point in time, we still have gaslimit + buffer gas available.

This property is not maintained because of the gap between the check and the execution.

The check is as follows, where `FINALIZE_GAS_BUFFER == 20_000`:

```
require(
    gasleft() >= _tx.gasLimit + FINALIZE_GAS_BUFFER,
    "OptimismPortal: insufficient gas to finalize withdrawal"
);
```

After this check, we know that the current execution context has at least 20,000 more gas than the gas limit. However, we then proceed to spend gas by (a) assigning the `l2Sender` storage variable, which uses 2900 gas because it's assigning from a non-zero value, and (b) perform some additional operations to prepare the contract for the external call.

The result is that, by the time the call is sent with `gasleft() - FINALIZE_GAS_BUFFER` gas, `gasleft()` is 5122 lower than it was in the initial check.

Mathematically, this can be expressed as:

- `gasAtCheck >= gasLimit + 20000`
- `gasSent == gasAtCall - 20000`
- `gasAtCall == gasAtCheck - 5122`

Rearranging, we get `gasSent >= gasLimit + 20000 - 5122 - 20000`, which simplifies to `gasSent >= gasLimit - 5122`.

Impact

For any withdrawal where a user sets their gas limit within 5122 of the actual gas their execution requires, a malicious user can call `finalizeWithdrawalTransaction()` on their behalf with enough gas to pass the check, but not enough for execution to succeed.

The result is that the withdrawing user will have their funds permanently locked in the `OptimismPortal` contract.

Proof of Concept

To test this behavior in a sandboxed environment, you can copy the following proof of concept.

Here are three simple contracts that replicate the behavior of the Portal, as well as an external contract that uses a predefined amount of gas.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

library SafeCall {
    /**
```

```
     * @notice Perform a low level call without copying any returndata
     *
     * @param _target   Address to call
     * @param _gas      Amount of gas to pass to the call
     * @param _value    Amount of value to pass to the call
     * @param _calldata Calldata to pass to the call
     */
    function call(
        address _target,
        uint256 _gas,
        uint256 _value,
        bytes memory _calldata
    ) internal returns (bool) {
        bool _success;
        assembly {
            _success := call(
                _gas, // gas
                _target, // recipient
                _value, // ether value
                add(_calldata, 0x20), // inloc
                mload(_calldata), // inlen
                0, // outloc
                0 // outlen
            )
        }
        return _success;
    }
}

contract GasUser {
    uint[] public s;

    function store(uint i) public {
        for (uint j = 0; j < i; j++) {
            s.push(1);
        }
    }
}

contract Portal {
    address l2Sender;

    struct Transaction {
        uint gasLimit;
        address sender;
        address target;
        uint value;
        bytes data;
```


```
    }

    constructor(address _l2Sender) {
        l2Sender = _l2Sender;
    }

    function execute(Transaction memory _tx) public {
        require(
            gasleft() >= _tx.gasLimit + 20000,
            "OptimismPortal: insufficient gas to finalize withdrawal"
        );

        // Set the l2Sender so contracts know who triggered this withdrawal on
↪   L2.
        l2Sender = _tx.sender;

        // Trigger the call to the target contract. We use SafeCall because we
↪   don't
        // care about the returndata and we don't want target contracts to be
↪   able to force this
        // call to run out of gas via a returndata bomb.
        bool success = SafeCall.call(
            _tx.target,
            gasleft() - 20000,
            _tx.value,
            _tx.data
        );
    }
}
```

Here is a Foundry test that calls the Portal with various gas values to expose this vulnerability:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/Portal.sol";

contract PortalGasTest is Test {
    Portal public c;
    GasUser public gu;

    function setUp() public {
        c = new Portal(0x000000000000000000000000000000000000dEaD);
        gu = new GasUser();
    }
```

```
    function testGasLimitForGU() public {
        gu.store{gas: 44_602}(1);
        assert(gu.s(0) == 1);
    }

    function _executePortalWithGivenGas(uint gas) public {
        c.execute{gas: gas}(Portal.Transaction({
            gasLimit: 44_602,
            sender: address(69),
            target: address(gu),
            value: 0,
            data: abi.encodeWithSignature("store(uint256)", 1)
        }));
    }

    function testPortalCatchesGasTooSmall() public {
        vm.expectRevert(bytes("OptimismPortal: insufficient gas to finalize
↪  withdrawal"));
        _executePortalWithGivenGas(65681);
    }

    function testPortalSucceedsWithEnoughGas() public {
        _executePortalWithGivenGas(70803);
        assert(gu.s(0) == 1);
    }

    function testPortalBugWithInBetweenGasLow() public {
        _executePortalWithGivenGas(65682);

        // It now reverts because the array has a length of 0.
        vm.expectRevert();
        gu.s(0);
    }

    function testPortalBugWithInBetweenGasHigh() public {
        _executePortalWithGivenGas(70802);

        // It now reverts because the array has a length of 0.
        vm.expectRevert();
        gu.s(0);
    }
}
```

Summarizing the results of this test:

- We verify that the call to the target contract succeeds with 44,602 gas, and set that as gasLimit for all tests.

- When we send 65,681 or less gas, it's little enough to be caught by the require statement.
- When we send 70,803 or more gas, the transaction will succeed.
- When we send any amount of gas between these two values, the require check is passed but the transaction fails.

Code Snippet

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L310-L329

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/libraries/SafeCall.sol#L17-L36

Tool used

Manual Review

Recommendation

Instead of using one value for `FINALIZE_GAS_BUFFER`, two separate values should be used that account for the gas used between the check and the call.

Discussion

rcstanciu

Comment from Optimism

Description: Malicious user can finalize other's withdrawal with less than specified gas limit, leading to loss of funds

Reason: This issue allows for attackers to **permissionlessly** brick a withdrawal transaction that specifies a minimum gas limit < `required_gas + 5122`. Most users will simply simulate the L1 transaction to identify the minimum gas needed for their withdrawal tx without including any padding, implying a high likelihood that this will affect a majority of our withdrawal transactions. Great find.

Action: In order to fix this issue, we need to factor in the gas used between the check on L313 of `OptimismPortal.sol` and the call to the destination on L324. Exploit PoC: here

Issue H-2: Withdrawals with high gas limits can be bricked by a malicious user, permanently locking funds

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/96

Found by

Allarious, obront, ladboy233, Barichek

Summary

Transactions to execute a withdrawal from the Optimism Portal require the caller to send enough gas to cover `gasLimit` specified by the withdrawer.

Because the EVM limits the total gas forwarded on to 63/64ths of the total `gasleft()` (and silently reduces it to this value if we try to send more) there are situations where transactions with high gas limits will be vulnerable to being reverted.

Because there are no replays on this contract, the result is that a malicious user can call `finalizeWithdrawalTransaction()` with a precise amount of gas, cause the withdrawer's withdrawal to fail, and permanently lock their funds.

Vulnerability Detail

Withdrawals can be withdrawn from L2's `L2ToL1MessagePasser` contract to L1's `OptimismPortal` contract. This is a less "user-friendly" withdrawal path, presumably for users who know what they are doing.

One of the quirks of the `OptimismPortal` is that there is no replaying of transactions. If a transaction fails, it will simply fail, and all ETH associated with it will remain in the `OptimismPortal` contract. Users have been warned of this and understand the risks, so Optimism takes no responsibility for user error.

In order to ensure that failed transactions can only happen at the fault of the user, the contract implements a check to ensure that the gasLimit is sufficient:

```
require(
    gasleft() >= _tx.gasLimit + FINALIZE_GAS_BUFFER,
    "OptimismPortal: insufficient gas to finalize withdrawal"
);
```

When the transaction is executed, the contract requests to send along all the remaining gas, minus the hardcoded `FINALIZE_GAS_BUFFER` for actions after the call. The goal is that this will ensure that the amount of gas forwarded on is at least the gas limit specified by the user.

Optimism is aware of the importance of this property being correct when they write in the comments:

> "We want to maintain the property that the amount of gas supplied to the call to the target contract is at least the gas limit specified by the user. We can do this by enforcing that, at this point in time, we still have gaslimit + buffer gas available."

The issue is that the EVM specifies the maximum gas that can be sent to an external call as 63/64ths of the `gasleft()`. For very large gas limits, this 1/64th that remains could be greater than the hardcoded FINALIZE_GAS_BUFFER value. In this case, less gas would be forwarded along than was directed by the contract.

Here is a quick overview of the math:

- We need X gas to be sent as a part of the call.
- This means we need `X * 64 / 63` gas to be available at the time the function is called.
- However, the only check is that we have `X + 20_000` gas a few operations prior to the call (which guarantees that we have `X + 14878` at the time of the call).
- For any situation where `X / 64 > 14878` (in other words, when the amount of gas sent is greater than `952_192`), the caller is able to send an amount of gas that passes the check, but doesn't forward the required amount on in the call.

Impact

For any withdrawal with a gas limit of at least 952,192, a malicious user can call `finalizeWithdrawalTransaction()` with an amount of gas that will pass the checks, but will end up forwarding along less gas than was specified by the user.

The result is that the withdrawing user can have their funds permanently locked in the `OptimismPortal` contract.

Proof of Concept

To test this behavior in a sandboxed environment, you can copy the following proof of concept.

Here are three simple contracts that replicate the behavior of the Portal, as well as an external contract that uses a predefined amount of gas.

(Note that we added 5122 to the gas included in the call to correct for the other bug we submitted, as this issue remains even when the other bug is patched.)

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;
```

```
library SafeCall {
    /**
     * @notice Perform a low level call without copying any returndata
     *
     * @param _target   Address to call
     * @param _gas      Amount of gas to pass to the call
     * @param _value    Amount of value to pass to the call
     * @param _calldata Calldata to pass to the call
     */
    function call(
        address _target,
        uint256 _gas,
        uint256 _value,
        bytes memory _calldata
    ) internal returns (bool) {
        bool _success;
        assembly {
            _success := call(
                _gas, // gas
                _target, // recipient
                _value, // ether value
                add(_calldata, 0x20), // inloc
                mload(_calldata), // inlen
                0, // outloc
                0 // outlen
            )
        }
        return _success;
    }
}

contract GasUser {
    uint[] public s;

    function store(uint i) public {
        for (uint j = 0; j < i; j++) {
            s.push(1);
        }
    }
}

contract Portal {
    address l2Sender;

    struct Transaction {
        uint gasLimit;
        address sender;
        address target;
```


```
        uint value;
        bytes data;
    }

    constructor(address _l2Sender) {
        l2Sender = _l2Sender;
    }

    function execute(Transaction memory _tx) public {
        require(
            gasleft() >= _tx.gasLimit + 20000,
            "OptimismPortal: insufficient gas to finalize withdrawal"
        );

        // Set the l2Sender so contracts know who triggered this withdrawal on
↪   L2.
        l2Sender = _tx.sender;

        // Trigger the call to the target contract. We use SafeCall because we
↪   don't
        // care about the returndata and we don't want target contracts to be
↪   able to force this
        // call to run out of gas via a returndata bomb.
        bool success = SafeCall.call(
            _tx.target,
            gasleft() - 20000 + 5122, // fix for other bug
            _tx.value,
            _tx.data
        );
    }
}
```

Here is a Foundry test that calls the Portal with various gas values to expose this vulnerability:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/Portal.sol";

contract PortalGasTest is Test {
    Portal public c;
    GasUser public gu;

    function setUp() public {
        c = new Portal(0x000000000000000000000000000000000000dEaD);
```

```
        gu = new GasUser();
    }

    function testGasLimitForGU() public {
        gu.store{gas: 11_245_655}(500);
        assert(gu.s(499) == 1);
    }

    function _executePortalWithGivenGas(uint gas) public {
        c.execute{gas: gas}(Portal.Transaction({
            gasLimit: 11_245_655,
            sender: address(69),
            target: address(gu),
            value: 0,
            data: abi.encodeWithSignature("store(uint256)", 500)
        }));
    }

    function testPortalCatchesGasTooSmall() public {
        vm.expectRevert(bytes("OptimismPortal: insufficient gas to finalize
↪   withdrawal"));
        _executePortalWithGivenGas(11_266_734);
    }

    function testPortalSucceedsWithEnoughGas() public {
        _executePortalWithGivenGas(11_433_180);
        assert(gu.s(499) == 1);
    }

    function testPortalBugWithInBetweenGasLow() public {
        _executePortalWithGivenGas(11_266_735);

        // It now reverts because the array has a length of 0.
        vm.expectRevert();
        gu.s(0);
    }

    function testPortalBugWithInBetweenGasHigh() public {
        _executePortalWithGivenGas(11_433_179);

        // It now reverts because the array has a length of 0.
        vm.expectRevert();
        gu.s(0);
    }
}
```

As you can see:

- We verify that the call to the target contract succeeds with 11,245,655 gas, and set that as gasLimit for all tests. This is the `X` from our formula above.
- This means that we need `11_245_655 * 64 / 63 = 11_424_157` gas available at the time the call is made.
- The test uses `9023` gas before it makes our call, so we can see that if we send `11_424_157 + 9_023 = 11_433_180` gas, the test passes.
- Similarly, if we send `11_266_734` gas, the total gas will be small enough to fail the require check.
- But in the sweet spot between these values, we have enough gas to pass the require check, but when we get to the call, the amount of gas requested is more than 63/64ths of the total, so the EVM sends less than we asked for. As a result, the transaction fails.

Code Snippet

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L310-L329

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/libraries/SafeCall.sol#L17-L36

Tool used

Manual Review

Recommendation

Change the check to account for this 63/64 rule:

```
require(
    gasleft() >= (_tx.gasLimit + FINALIZE_GAS_BUFFER) * 64 / 63,
    "OptimismPortal: insufficient gas to finalize withdrawal"
);
```

Discussion

rcstanciu

Comment from Optimism

Description: Withdrawals with high gas limits can be bricked by a malicious user, permanently locking funds

Reason: The issue is that we do not account for the 63/64ths rule specified in EIP-150 when checking the remaining gas available for executing a withdrawal transaction. This means that a malicious user can **permissionlessly** brick a withdrawal that has a minimum gas limit > 952,192 by forcing the `OptimismPortal` to forward an incorrect amount of gas, causing the withdrawal transaction to fail unexpectedly.

Action: In order to fix this issue, we need to factor in the 63/64 rule specified in EIP-150 when checking the remaining gas available for executing a withdrawal transaction on L313 of the `OptimismPortal` contract. See Zach's suggestion. (**VERY IMPORTANT NOTE**: This issue exists independently of #109.)

Issue H-3: Causing users lose their fund during finalizing withdrawal transaction

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/87

Found by

HE1M

Summary

A malicious user can make users lose their fund during finalizing their withdrawal. This is possible due to presence of reentrancy guard on the function `relayMessage`.

Vulnerability Detail

- Bob (a malicious user) creates a contract (called `AttackContract`) on L1.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.0;

struct WithdrawalTransaction {
    uint256 nonce;
    address sender;
    address target;
    uint256 value;
    uint256 gasLimit;
    bytes data;
}

interface IOptimismPortal {
    function finalizeWithdrawalTransaction(WithdrawalTransaction memory _tx)
        external;
}

contract AttackContract {
    bool public donotRevert;
    bytes metaData;
    address optimismPortalAddress;

    constructor(address _optimismPortal) {
        optimismPortalAddress = _optimismPortal;
    }

    function enableRevert() public {
        donotRevert = true;
```

```
    }

    function setMetaData(WithdrawalTransaction memory _tx) public {
        metaData = abi.encodeWithSelector(
            IOptimismPortal.finalizeWithdrawalTransaction.selector,
            _tx
        );
    }

    function attack() public {
        if (!donotRevert) {
            revert();
        } else {
            optimismPortalAddress.call(metaData);
        }
    }
}
```

- Bob sends a message from L2 to L1 by calling the function `sendMessage` with the following parameters. He intends to call the function `attack()` through relaying the message from L2 to L1.
  - `_target` = address of `AttackContract` on L1
  - `_message` = abi.encodeWithSignature("attack()")
  - `_minGasLimit` = just big enough so that the transaction can be executed https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L212
- On the L1 side, after challenge period and validation elapsed, the function `attack()` on contract `AttackContract` will be called during relaying the message. https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L324
- But, since `donotRevert` is `false` in the contract `AttackContract`, the relayed message will be unsuccessful. So, we will have `failedMessages[versionedHash] = true`. It means that it is possible again to retry relaying the message later. https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L331

```
if (!donotRevert) {
    revert();
}
```

- Then, Bob calls the function `enableRevert` to set `donotRevert` to `true`. So that if later the function `attack()` is called again, it will not revert.

```
function enableRevert() public {
    donotRevert = true;
}
```

- Then, Bob notices that Alice is withdrawing large amount of fund from L2 to L1. Her withdrawal transaction is proved but she is waiting for the challenge period to be finished to finalize it.
- Then, Bob calls the function `setMetaData` on the contract `AttackContract` with the following parameter:
  - `_tx` = Alice's withdrawal transaction
- By doing so, the `metaData` will be equal to `finalizeWithdrawalTransaction.selector` + Alice's withdrawal transaction.

```
function setMetaData(WithdrawalTransaction memory _tx) public {
     metaData = abi.encodeWithSelector(
         IOptimismPortal.finalizeWithdrawalTransaction.selector,
         _tx
     );
 }
```

- Now, after the challenge period is passed, and before the function `finalizeWithdrawalTransaction` is called by anyone (Alice), Bob calls the function `relayMessage` with the required data to retry his previous failed message again.
- This time, since `donotRevert` is `true`, the call to function `attack()` will not revert, instead the body of `else clause` will be executed.

```
else {
    optimismPortalAddress.call(metaData);
}
```

- In the `else clause`, it calls the function `finalizeWithdrawalTransaction` with Alice's withdrawal transaction as the parameter, to finalize Alice's transaction. https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L243
- During finalizing Alice's withdrawal transaction, everything goes smoothly (as the challenge period is passed, and everything is valid) until the call to the function `relayMessage` in the contract `CrossDomainMessanger`. https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L324

- Due to the reentrancy guard, the call will be unsuccessful. https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L263
- Please note that the flow is as follows: Bob ==> `CrossDomainMessenger.relayMessage` ==> `AttackContract.attack` ==> `OptimismPortal.finalizeWithdrawalTransaction` => `CrossDomainMessenger.relayMessage`
- Since, the failed call is not handled during finalizing the message, the transaction will be finished without any error.
- Then, Bob's relayed message transaction will be finished successfully.
- By doing so, Alice's withdrawal transaction is flagged as finalized, but in reality it was not because of reentrancy guard. So, Alice loses her fund. https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L308

In summary the attack is as follows:

1. Bob creates a malicious contract on L1 called `AttackContract`.
2. Bob sends a message from L2 to L1 to call the function `AttackContract.attack` on L1.
3. On L1 side, after the challenge period is passed, the function `AttackContract.attack` will be called.
4. Message relay on L1 will be unsuccessful, because the function `AttackContract.attack` reverts. So, Bob's message will be flagged as failed message.
5. Bob sets `AttackContract.donotRevert` to true.
6. Bob waits for an innocent user to request withdrawal transaction.
7. Bob waits for the innocent user's withdrawal transaction to be proved.
8. Bob sets meta data in his malicious contract based on the innocent user's withdrawal transaction.
9. Bob waits for the challenge period to be passed.
10. After the challenge period is elapsed, Bob retries to relay his failed message again.
11. `CrossDomainMessenger.relayMessage` will call the `AttackContract.attack`, then it calls `OptimismPortal.finalizeWithdrawalTransaction` to finalize innocent user's withdrawal transaction. Then, it calls `CrossDomainMessenger.relayMessage`, but it will be unsuccessful because of reentrancy guard.

12. After finalizing the innocent user's withdrawal transaction, Bob's message will be flagged as successful.
13. So, innocent user's withdrawal transaction is flagged as finalized, while it is not.

Impact

By doing this attack it is possible to prevent users from withdrawing their fund. Moreover, they lose their fund because withdrawal is flagged as finalized, but the withdrawal sent to `L1CrossDomainMessanger` was not successful.

Code Snippet

Tool used

Manual Review

Recommendation

Maybe it is better to use the following code instead of: https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L324-L329

```
try IL1CrossDomainMessanger.relayMessage(...) {} catch Error(string memory
↪  reason) {
    if (
        keccak256(abi.encodePacked(reason)) ==
        keccak256(abi.encodePacked("ReentrancyGuard: reentrant call"))
    ) {
        revert("finalizing should be reverted");
    }
}
```

Discussion

rcstanciu

Comment from Optimism

Description: Causing users lose their fund during finalizing withdrawal transaction

Reason: This is a very creative exploit that takes advantage of the `L1CrossDomainMessenger`'s reentrancy guard on `relayMessage`. By creating an attack contract that reverts the first time a message is relayed by the `OptimismPortal` to the `L1CrossDomainMessenger`, but can be toggled to call

`finalizeWithdrawalTransaction` for a **different** withdrawal transaction when the attacker replays it via the `L1CrossDomainMessenger`, the attacker can force the honest withdrawal's call to the `L1CrossDomainMessenger` to revert, effectively bricking it. PoC: here

Action: If a withdrawal transaction fails, we should check to see if the `_target` of the `WithdrawalTransaction` is the `L1CrossDomainMessenger`. If so, check if the call's revert data is the reentrancy guard message. If it is, we should revert the `finalizeWithdrawalTransaction` message and *NOT* mark the withdrawal as finalized.

Issue M-1: Batcher frames are incorrectly decoded leading to consensus split

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/279

Found by

obront

Summary

There is an error in the implementation of how frames are decoded, which will allow invalid frames to be accepted. This can allow a malicious sequencer to cause a consensus split between op-node implementations and the (correct) reference implementation.

Vulnerability Detail

Optimism implements a highly efficient derivation scheme based on compression of L2 data that is sent to an L1 address. The spec clearly defines how this works. Channels are split into frames, each one encoded as defined here. Frames will be aggregated by the derivation driver.

Docs state that:

> All data in a frame is fixed-size, except the frame_data. The fixed overhead is 16 + 2 + 4 + 1 = 23 bytes. Fixed-size frame metadata avoids a circular dependency with the target total data length, to simplify packing of frames with varying content length.

Specifically:

> is_last is a single byte with a value of 1 if the frame is the last in the channel, 0 if there are frames in the channel. Any other value makes the frame invalid (it must be ignored by the rollup node).

Clearly, `is_last` is mandatory as per the specs. However, if we look at the code it will accept a frame even if `is_last` is not supplied.

Decoding of the frame is done in `frame.go`, in the `UnmarshalBinary` function. After reading the frame data, only the last byte remains.

```
if isLastByte, err := r.ReadByte(); err != nil && err != io.EOF {
    return fmt.Errorf("error reading final byte: %w", err)
} else if isLastByte == 0 {
    f.IsLast = false
    return err
```


SHERLOCK

```
} else if isLastByte == 1 {
    f.IsLast = true
    return err
} else {
    return errors.New("invalid byte as is_last")
}
```

If the `ByteReader` object is empty, reading the next byte will return an `EOF` and the error clause is skipped. The result of `ReadByte` when an error occurs is `undefined`, however in all Go setups we've tested `isLastByte` is zero. This means it sets `f.IsLast = false` and returns the `EOF`.

Back in `ParseFrames` which calls `UnmarshalBinary`, the `EOF` is ignored and the frame is accepted:

```
for buf.Len() > 0 {
    var f Frame
    if err := (&f).UnmarshalBinary(buf); err != io.EOF && err != nil {
        return nil, err
    }
    frames = append(frames, f)
}
```

So, it is demonstrated that an invalid frame is accepted by the Optimism implementation, provided the frame is the last one in the frames buffer. The impact is that a malicious sequencer can cause a consensus split between correct implementations and the reference implementation. It has been defined by the rules as Medium severity:

```
- Causing a consensus failure
- Explanation: There is one sequencer op-node submitting transaction batches to
↪  L1, but many verifier op-nodes will read these batches and check the results
↪  of its execution. The sequencer and verifiers must remain in consensus, even
↪  in the event of an L1 reorg.
```

All that is needed is to send in different frame packages two frames of a channel, omit the `is_last` byte in the first frame and make sure it is the last frame in the package.

Impact

Malicious sequencer can easily cause a consensus split by taking advantage of the incorrect frame reading logic in op-node.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/407f97b9d13448b766624995ec824d3059d4d4f6/op-node/rollup/derive/frame.go#L93

Tool used

Manual Review

Recommendation

In `UnmarshalBinary`, return a non-EOF error when `is_last` byte does not exist.

Discussion

rcstanciu

Comment from Optimism

Description: A malicious or buggy batcher could submit a frame that the op-node would accept that a spec compliant node would not due to a bug in our parsing code.

Reason: This would cause a consensus failure between different clients.

Action: We need to fix this code.

zobront

Escalate for 250 USDC

This submission concerns a specific consensus failure due to parsing of batcher frames.

It has been wrongly duped to #53, which concerns a contract level issue with posted output roots for withdrawals.

Optimism has acknowledged the validity of the issue and this is certainly just a misclick, as the issues are entirely unrelated.

sherlock-admin

> Escalate for 250 USDC
>
> This submission concerns a specific consensus failure due to parsing of batcher frames.
>
> It has been wrongly duped to #53, which concerns a contract level issue with posted output roots for withdrawals.

> Optimism has acknowledged the validity of the issue and this is certainly just a misclick, as the issues are entirely unrelated.

You've created a valid escalation for 250 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Evert0x

Escalation accepted and resolving incorrect duplication state

sherlock-admin

> Escalation accepted and resolving incorrect duplication state

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

Issue M-2: Censorship resistance is undermined and bridging of assets can be DOSed at low cost

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/277

Found by

obront, Robert, cergyk

Summary

All L1->L2 transactions go through OptimismPortal's `depositTransaction` function. It is wrapped through the `metered` modifier. The goal is to create a gas market for L1->L2 transactions and not allow L1 TXs to fill up L2 batches (as the gas for deposit TX in L2 is payed for by the system), but the mechanism used makes it too inexpensive for a malicious user to DOS and censor deposits.

Vulnerability Detail

It is possible for a malicious actor to snipe arbitrary L1->L2 transactions in the mempool for far too cheaply. This introduces two impacts:

1. Undermines censorship resistance guarantees by Optimism
2. Griefs users who simply want to bridge assets to L2

The core issue is the check in ResourceMetering.sol:

```
// Make sure we can actually buy the resource amount requested by the user.
params.prevBoughtGas += _amount;
require(
    int256(uint256(params.prevBoughtGas)) <= MAX_RESOURCE_LIMIT,
    "ResourceMetering: cannot buy more gas than available gas limit"
);
```

Note that `params.prevBoughtGas` is reset per block. This means attacker can view a TX in the mempool and wrap up the following flashbot bundle:

1. Attacker TX to `depositTransaction`, with gasLimit = 8M (MAX_RESOURCE_LIMIT)
2. Victim TX to `depositTransaction`

The result is that attacker's transaction will execute and victim's TX would revert. It is unknown how this affects the UI and whether victim would be able to resubmit this TX again easily, but regardless it's clearly griefing user's attempt to bridge an asset. Note that a reverted TX is different from an uncompleted TX from a UX point of view.

From a censorship resistance perspective, there is nothing inherently preventing attack to continually use this technique to block out all TXs, albert gas metering price will rise as will be discussed.

Now we can demonstrate the cost of the attack to be low. Gas burned by the modifier is calculated as:

```
// Determine the amount of ETH to be paid.
uint256 resourceCost = _amount * params.prevBaseFee;
...
uint256 gasCost = resourceCost / Math.max(block.basefee, 1000000000);
```

`params.prevBaseFee` is initialized at 1e9 and goes up per block by a factor of 1.375 when gas market is drained, while going down by 0.875 when gas market wasn't used at all.

If we take the initial value, `resourceCost = 8e6 * 1e9 = 8e15`. If we assume tip is negligible to `block.basefee`, L1 gas cost in ETH equals `resourceCost` (divide by basefee and multiply by basefee). Therefore, cost of this snipe TX is:

`8e15 / 1e18 (ETH decimals) * 1600 (curr ETH price) = $12.80`

The result is an extremely low price to pay, and even taking into account extra tips for frontrunning, is easily achievable.

In practice `prevBaseFee` will represent the market price for L2 gas. If it goes lower than initial value, DOSing will become cheaper, while if it goes higher it will become more expensive. The key problem is that the attacker's cost is too similar to the victim's cost. If victim is trying to pass a 400k TX, attacker needs to buy a 7.6M of gas. This gap is too small and the resulting situation is that for DOS to be too expensive for attacker, TX would have to be far too expensive for the average user.

Impact

Censorship resistance is undermined and bridging of assets can be DOSed at low cost.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/407f97b9d13448b766624995ec824d3059d4d4f6/packages/contracts-bedrock/contracts/L1/ResourceMetering.sol#L133

Tool used

Manual Review

Recommendation

It is admittedly difficult to balance the need for censorship resistance with the prevention of L2 flooding via L1 TXs. However, the current solution which will make a victim's TX revert at hacker's will is inadequate and will lead to severe UX issues for users.

Recommendation would be to not require gas spending to be under `MAX_RESOURCE_LIMIT` and solve it in L2 sequencing. It's already close enough to the L1 gas limit so that it can't be really abused.

Discussion

rcstanciu

Comment from Optimism

Description: Deposit griefing by front running and filling up the MAX_RESOURCE_LIMIT

Reason: DOS

Issue M-3: Panic when decoding a malformed deposit transaction JSON string

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/276

Found by

shw

Summary

When decoding a deposit transaction JSON string without the `"gas"` field, a panic/runtime error is triggered due to a nil pointer dereference.

Vulnerability Detail

The `op-geth/core/types/transaction_marshalling.go` file defines how transactions are encoded and decoded from JSON format. In the `UnmarshalJSON()` function, from L283 to L315, a new logic is added for the deposit transaction type, `DepositTxType`. The bug happens at L293, where the `dec.Gas` field is dereferenced without checking it against `nil` first. As a result, if the provided deposit transaction JSON string does not have the `"gas"` field, the nil pointer dereference will trigger a runtime error and crash the program.

For a PoC: Add the following test case to `op-geth/core/types/transaction_test.go` and run `cd op-geth && go test -run TestDecodeJSON -v ./core/types`

```
func TestDecodeJSON(t *testing.T) {
    // the "gas" field does not exist
    var data =
↪   []byte("{\"type\":\"0x7e\",\"nonce\":null,\"gasPrice\":null,\"maxPriorityFee
↪   PerGas\":null,\"maxFeePerGas\":null,\"value\":\"0x1\",\"input\":\"0x61626364
↪   6566\",\"v\":null,\"r\":null,\"s\":null,\"to\":null,\"sourceHash\":\"0x00000
↪   0000000000000000000000000000000000000000000000000000000000\",\"from\":\"0x0
↪   0000000000000000000000000000000000001\",\"isSystemTx\":false,\"hash\":\"0x
↪   a4341f3db4363b7ca269a8538bd027b2f8784f84454ca917668642d5f6dffdf9\"}")
    var parsedTx = &Transaction{}
    _ = json.Unmarshal(data, &parsedTx) // will panic here
}
```

Impact

A possible exploit scenario is targeting an `ethclient` compiled based on `op-geth`. For example, according to `op-geth/ethclient/ethclient.go`, the `BlockByHash()` API makes a RPC `eth_getBlockByHash` request to an RPC endpoint and expect to receive

SHERLOCK

a `json.RawMessage` data that represents the queried block. In the `getBlock()` function, the JSON raw message is then decoded into a `rpcBlock`, containing a list of `rpcTransaction`s with type of `Transaction`. Therefore, a malicious RPC endpoint can construct a deposit transaction without the `"gas"` field in a block and return the block to the `ethclient`. The `ethclient` will fail to decode the received block and crash due to this bug.

Marking this issue as medium severity according to previous similar audit findings from Sigma Prime.

Code Snippet

Please refer to `op-geth/core/types/transaction_marshalling.go#L293`, `op-geth/ethclient/ethclient.go#L77-L79` and `op-geth/ethclient/ethclient.go#L118-L126`.
https://github.com/ethereum-optimism/op-geth/blob/985086bf2a5c61e76a8ce7c74ac029660751e260/core/types/transaction_marshalling.go#L293

Tool used

Manual Review

Recommendation

Check whether `dec.Gas` is `nil` before dereferencing it:

```
if dec.Gas == nil {
    return errors.New("missing required field 'gas' in transaction")
}
```

Discussion

rcstanciu

Comment from Optimism

Description: Panic in transaction unmarshalling code

Reason: This is a legit bug in our transaction unmarshalling code.

Action: Check for nil gas before unmarshalling

SHERLOCK

Issue M-4: [High] Function MigrateWithdrawal() may set gas limit so high for old withdrawals when migrating them by mistake and they can't be relayed in the L1 and users funds would be lost

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/235

Found by

unforgiven

Summary

Function `MigrateWithdrawal()` in migrate.go will turn a LegacyWithdrawal into a bedrock style Withdrawal. it should set a min gas limit value for the withdrawals. to calculate a gas limit contract overestimates it and if the value goes higher than L1 maximum gas in the block then the withdraw can't be relayed in the L1 and users funds would be lost while the withdraw could be possible before the migration it won't be possible after it.

Vulnerability Detail

This is `MigrateWithdrawal()` code:

```
// MigrateWithdrawal will turn a LegacyWithdrawal into a bedrock
// style Withdrawal.
func MigrateWithdrawal(withdrawal *LegacyWithdrawal, l1CrossDomainMessenger
↪  *common.Address) (*Withdrawal, error) {
    // Attempt to parse the value
    value, err := withdrawal.Value()
    if err != nil {
        return nil, fmt.Errorf("cannot migrate withdrawal: %w", err)
    }

    abi, err := bindings.L1CrossDomainMessengerMetaData.GetAbi()
    if err != nil {
        return nil, err
    }

    // Migrated withdrawals are specified as version 0. Both the
    // L2ToL1MessagePasser and the CrossDomainMessenger use the same
    // versioning scheme. Both should be set to version 0
    versionedNonce := EncodeVersionedNonce(withdrawal.Nonce, new(big.Int))
    // Encode the call to `relayMessage` on the `CrossDomainMessenger`.
    // The minGasLimit can safely be 0 here.
```


SHERLOCK

```
    data, err := abi.Pack(
        "relayMessage",
        versionedNonce,
        withdrawal.Sender,
        withdrawal.Target,
        value,
        new(big.Int),
        withdrawal.Data,
    )
    if err != nil {
        return nil, fmt.Errorf("cannot abi encode relayMessage: %w", err)
    }

    // Set the outer gas limit. This cannot be zero
    gasLimit := uint64(len(data)*16 + 200_000)

    w := NewWithdrawal(
        versionedNonce,
        &predeploys.L2CrossDomainMessengerAddr,
        l1CrossDomainMessenger,
        value,
        new(big.Int).SetUint64(gasLimit),
        data,
    )
    return w, nil
}
```

As you can see it sets the gas limit as `gasLimit := uint64(len(data)*16 + 200_000)` and contract set 16 gas per data byte but in Ethereum when data byte is 0 then the overhead intrinsic gas is 4 and contract overestimate the gas limit by setting 16 gas for each data. this can cause messages with big data(which calculated gas is higher than 30M) to not be relay able in the L1 because if transaction gas set lower than calculated gas then OptimisimPortal would reject it and if gas set higher than calculated gas then miners would reject the transaction. while if code correctly estimated the required gas the gas limit could be lower by the factor of 4. for example a message with about 2M zeros would get gas limit higher than 30M and it won't be withdrawable in the L1 while the real gas limit is 8M which is relayable.

Impact

some withdraw messages from L2 to L1 that could be relayed before the migration can't be relayed after the migration because of the wrong gas estimation.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/3f4b3c328153a8aa03611158b6984d624b17c1d9/op-chain-ops/crossdomain/migrate.go#L83-L88

Tool used

Manual Review

Recommendation

calculate gas estimation correctly, 4 for 0 bytes and 16 for none zero bytes.

Discussion

rcstanciu

Comment from Optimism

Description: Incorrect gas limit calculation for migrated withdrawals

Reason: This is a legitimate issue.

Action: Implement correct gas estimation

0xunforgiven

Escalate for 111 USDC

This should labeled as High because some of old legit withdrawals(L2 -> L1) won't be executed in the L1 after the bedrock migration. Those withdrawals belongs to L2CrossDomainMessages which promised to be delivered in L1.

Users may lose funds and there is no limit for it. there may be users who wants to withdraw 1M funds with L2StandardBridge and because of this issue their funds would be lost even so the L2StandardBridge is supposed to deliver the funds! The withdrawals with large amount of data isn't that rare, There may be other protocols or users in L2 which uses the bridge to store data in the L1 and send withdrawals with large data to the L2CrossDomainMessenger and because of the issue their withdrawal message won't be finalized in L1 after migration.

It is true that this would only happen to some of the withdrawal messages but as we have other gas related issues that blocks withdrawal of limited number of withdrawals(#96 which is only happen to withdrawals that would revert if gas was 5K lower and #109 which happened to withdrawals with higher than ~1M gas) and they considered as High so to be fair This issue should be High too.

This is direct fund loss if it happens to user ETH bridges and also cause L2CrossDomainMessenger's messages to not finalized and be replayable in L1 if it happens to it's withdrawal messages.

sherlock-admin

> Escalate for 111 USDC
>
> This should labeled as High because some of old legit withdrawals(L2 -> L1) won't be executed in the L1 after the bedrock migration. Those withdrawals belongs to L2CrossDomainMessages which promised to be delivered in L1.
>
> Users may lose funds and there is no limit for it. there may be users who wants to withdraw 1M funds with L2StandardBridge and because of this issue their funds would be lost even so the L2StandardBridge is supposed to deliver the funds! The withdrawals with large amount of data isn't that rare, There may be other protocols or users in L2 which uses the bridge to store data in the L1 and send withdrawals with large data to the L2CrossDomainMessenger and because of the issue their withdrawal message won't be finalized in L1 after migration.
>
> It is true that this would only happen to some of the withdrawal messages but as we have other gas related issues that blocks withdrawal of limited number of withdrawals(#96 which is only happen to withdrawals that would revert if gas was 5K lower and #109 which happened to withdrawals with higher than ~1M gas) and they considered as High so to be fair This issue should be High too.
>
> This is direct fund loss if it happens to user ETH bridges and also cause L2CrossDomainMessenger's messages to not finalized and be replayable in L1 if it happens to it's withdrawal messages.

You've created a valid escalation for 111 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Evert0x

Escalation rejected as there is insufficient evidence for high severity.

sherlock-admin

> Escalation rejected as there is insufficient evidence for high severity.

This issue's escalations have been rejected!

Watsons who escalated this issue will have their escalation amount deducted from their next payout.

Issue M-5: contract with only `IOptimismMintableERC20` interface is not compatible with `StandardBridge`

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/220

Found by

Barichek, lemonmon

Summary

If a custom contract implements only the `IOptimismMintableERC20`, but no the `ILegacyMintableERC20`, the contract is not compatible with the `StandardBridge`, as the bridge uses the `l1Token` function from the legacy interface

Vulnerability Detail

The comment in the `IOptimismMintableERC20` suggests that one can make a custom implementation of `OptimismMintableERC20` using the interface `IOptimismMintableERC20`.

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/IOptimismMintableERC20.sol#L8-L10

Also, the `StandardBridge`, which uses the `OptimismMintableERC20` has `_isOptimismMintableERC20` function, which checks whether the given token address is implementing `OptimismMintableERC20`. The function will be true if either of `ILegacyMintableERC20` or `IOptimismMintableERC20` is implemented. it means that if a token implements only one of the interfaces, it will return true.

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/StandardBridge.sol#L446-L450

However, if the given token passes the `_isOptimismMintableERC20`, the legacy function `l1Token` will be called on the token. If the token does not implement the legacy interface, the call will fail.

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L2/L2StandardBridge.sol#L170

Therefore, the token which only implements `IOptimismMintableERC20`, but not the `ILegacyMintableERC20`, is not compatible with `StandardBridge`.

Impact

Any custom contract without `l1Token` function will not be compatible with `StandardBridge`

Code Snippet

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/IOptimismMintableERC20.sol#L8-L10

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/StandardBridge.sol#L446-L450

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L2/L2StandardBridge.sol#L170

Tool used

Manual Review

Recommendation

It is unclear it is intended behavior. If the `_isOptimismMintableERC20` function returns true only when the both of interfaces are implemented, the token with only the `IOptimismMintableERC20` will be treated as if they are not the optimism mintable function, without failing.

Issue M-6: Crafted p2p spam can render nodes permanently unable to process L2 blocks

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/177

Found by

usmannk

Summary

The constraints in `BuildBlocksValidator` are meant to prevent p2p spam from bogging down the network. However they are not sufficient for stopping spam, but are strong enough to render nodes unable to catch up.

Vulnerability Detail

The `p2p.BuildBlocksValidator` function has several constraints that must be met before a gossiped block is considered valid. If a block is considered valid then it is gossiped to peers and ingested into the current node.

The constraints on a gossiped block `block` are, in order:

- `block` must be <10MB
- `block` must be compressed as valid snappy data
- `block` must be signed by the sequencer
- `block` must be a valid SSZ encoded block
- `block`.timestamp must within 60 seconds in the past and 5 seconds in the future
- `block.BlockHash` must be correctly calculated
- `block.BlockNumber` must not be in the set of the previous 100 unique block numbers that reached this step

These constraints are applied here:

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/op-node/p2p/gossip.go#L208

However, it is often the case that there are over 100 valid L2 blocks emitted per minute. For example, the 253 blocks between block 71404110 and 71404363 were emitted within 60 seconds.

In this example, an attacker would take the 200 blocks from 71404110-71404310 and replay them in order to a node for 60 seconds. Every block would be valid

because they each pass all constraints and the 100 block LRU cache for marking `seen` blocks would be evicted twice over during the process.

After 60 seconds, block `71404310 + 1` would be invalid to this node because its timestamp would be too old. At this point, for any new L2 block the node would either:

- mark it as invalid because it is too old
- or ingest it and throw it away because it is waiting for block `71404311`

As a result, the node will no longer be able to successfully process any L2 blocks from the p2p network.

Furthermore, because all 200 blocks are marked as valid several times, the node would gossip all of these blocks (potentially several times) to its peers who would suffer the same effect and impact their peers.

Impact

Attackers may halt targeted nodes.

Further knock-on potential halting of the node's peers, peers of peers, and so on.

Code Snippet

Tool used

Manual Review

Recommendation

Increase the size of the `blockHeightLRU` cache to more effectively catch and reject previously seen nodes. It would be reasonable to set it to 1000 instead of 100. Even at 1000 the cache should occupy less than 500kb in memory.

Discussion

rcstanciu

Comment from Optimism

Description: P2P buffer is too small

Reason: This is a valid attack, wherein the attacker replays blocks and eclipses a node from the P2P network.

Action: Increase buffer size

Issue M-7: LES (Light Ethereum Subprotocol) doesn't forward the transaction to the sequencer

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/175

Found by

Koolex

Summary

LES (Light Ethereum Subprotocol) doesn't forward the transaction to the sequencer when receiving it over RPC.

Vulnerability Detail

When a user submits a transaction to op-geth node (validator/verfier mode), the node sends the transaction to the sequencer, if no error, it adds it to the tx pool.

```
func (b *EthAPIBackend) SendTx(ctx context.Context, tx *types.Transaction) error
↪  {
    if b.eth.seqRPCService != nil {
        data, err := tx.MarshalBinary()
        if err != nil {
            return err
        }
        if err := b.eth.seqRPCService.CallContext(ctx, nil,
        ↪  "eth_sendRawTransaction", hexutil.Encode(data)); err != nil {
            return err
        }
    }
    return b.eth.txPool.AddLocal(tx)
}
```

https://github.com/ethereum-optimism/op-geth/blob/optimism-history/eth/api_backend.go#L253-L264

However, when LES, It only adds the transaction to the tx pool.

```
func (b *LesApiBackend) SendTx(ctx context.Context, signedTx *types.Transaction)
↪  error {
    return b.eth.txPool.Add(ctx, signedTx)
}
```

https://github.com/ethereum-optimism/op-geth/blob/optimism-history/les/api_backend.go#L193-L195

SHERLOCK

Note: Sequencer http flag is configured only if we're running in verifier mode.

Impact

- Transction isn't sent to the sequencer and will never be processed (submitted to L1).
- Inconsistency among op-geth nodes validators/verifiers and the sequencer.
- Additionally, from UX perspective, it is misleading as the user would think the transaction was submitted "successfully".

Code Snippet

Check above.

Tool used

Manual Review

Recommendation

Match this RPC change in the LES RPC. As it seems to be overlooked.

Ref: https://op-geth.optimism.io/

Discussion

rcstanciu

Comment from Optimism

Description: LES doesn't forward transactions to the sequencer

Reason: We don't support LES, so this is out-of-scope.

Action: No action.

koolexcrypto

Escalate for 53 USDC

I kindly ask you to reconsider the assessment due to the following reasons:

1. In the contest description, it lists the components that are in scope:

   > The key components of the system can be found in our monorepo at commit 3f4b3c3281. L1 Contracts L2 Contracts (AKA Predeploys) op-node op-geth (in its own repo)

op-geth is listed as it is, and it is never mentioned anywhere in the docs that LES is out of scope. As a participant, I spent quite some time to look for bugs in LES. otherwise, I would have shifted my focus on some other parts of the component considering the short time of the contest. On a personal level, it feels a bit unfair.

2. If you go over LES directory in the source code. You could see the code was adapted to match Optimism specs. This actually gave me a confirmation that it is in scope. Here are some examples:
    - L1CostFunc was added
        - https://github.com/ethereum-optimism/op-geth/blob/optimism/les/api_backend.go#L193
    - RollupSequencerHTTP and RollupHistoricalRPC
        - https://github.com/ethereum-optimism/op-geth/blob/optimism/les/client.go#L204-L222
    - Adapting stateAtTransaction func to consider L1CostFunc
        - https://github.com/ethereum-optimism/op-geth/blob/optimism/les/state_accessor.go#L70

sherlock-admin

Escalate for 53 USDC

I kindly ask you to reconsider the assessment due to the following reasons:

1. In the contest description, it lists the components that are in scope:

    The key components of the system can be found in our monorepo at commit 3f4b3c3281. L1 Contracts L2 Contracts (AKA Predeploys) op-node op-geth (in its own repo)

op-geth is listed as it is, and it is never mentioned anywhere in the docs that LES is out of scope. As a participant, I spent quite some time to look for bugs in LES. otherwise, I would have shifted my focus on some other parts of the component considering the short time of the contest. On a personal level, it feels a bit unfair.

2. If you go over LES directory in the source code. You could see the code was adapted to match Optimism specs. This actually gave me a confirmation that it is in scope. Here are some examples:

- L1CostFunc was added
    - https://github.com/ethereum-optimism/op-geth/blob/optimism/les/api_backend.go#L193

- RollupSequencerHTTP and RollupHistoricalRPC
  - https://github.com/ethereum-optimism/op-geth/blob/optimism/les/client.go#L204-L222
- Adapting stateAtTransaction func to consider L1CostFunc
  - https://github.com/ethereum-optimism/op-geth/blob/optimism/les/state_accessor.go#L70

You've created a valid escalation for 53 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Evert0x

Escalation accepted.

Labeling as medium severity.

This was an oversight on Optimism's part and there are markers that would suggest it should be in scope.

sherlock-admin

> Escalation accepted.
>
> Labeling as medium severity.
>
> This was an oversight on Optimism's part and there are markers that would suggest it should be in scope.

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

Issue M-8: Optimism Portal can run out of gas due to incorrect overhead estimation

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/138

Found by

GalloDaSballo

Summary

In contrast to `CrossDomainMessenger` which has a 5k gas buffer, the Optimism Portal doesn't, meaning all its relayed calls will have 5k+ less gas than intended.

This forces integrations (e.g. Bridges) to spend more gas by default, because of a logic flaw.

For this reason am filing the finding as Medium Severity:

- Programming Mistake (Math is incorrect)
- Call forwards less gas than intended and can revert because of it

Vulnerability Detail

`CrossDomainMessenger` will compute the gas-required, adding a 5k gas buffer https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L318-L324

It will then pass the remainder of the gas, minus the buffer as it's assumed to have been spent for the SSTORE

Optimism Portal on the other hand will not do that

Impact

By only checking that

```
gasleft() >= _tx.gasLimit + FINALIZE_GAS_BUFFER /// @audit At least gasLimit +
↪  buffer
```

The check will ensure that before the SSTORE + Call the buffer is available

However, the following SSTORE will consider 5k+ (5k for SSTORE, hundreds of gas for overhead)

This will leave the SafeCall with less gas than intended

```
gasleft() - FINALIZE_GAS_BUFFER, /// @audit gasLeft will be < tx.gasLimit
↪  because we've subtracted the same constant
```

`SafeCall` will have less gas than intended, meaning that every integrator that estimates their TXs accurately will have their tx revert.

Code Snippet

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L310-L328

Tool used

Manual Review

Recommendation

Recompute the buffer to add the extra 5k + the overhead of the SSTORE (in the few hundreds of gas)

To ensure the call has enough gas, you may also consider swapping positions between the SSTORE and the require

```
// Set the l2Sender so contracts know who triggered this withdrawal on L2.
l2Sender = _tx.sender;

require(
    gasleft() >= _tx.gasLimit + FINALIZE_GAS_BUFFER,
    "OptimismPortal: insufficient gas to finalize withdrawal"
);

// Trigger the call to the target contract. We use SafeCall because we don't
// care about the returndata and we don't want target contracts to be able to
↪  force this
// call to run out of gas via a returndata bomb.
bool success = SafeCall.call(
    _tx.target,
    gasleft() - FINALIZE_GAS_BUFFER,
    _tx.value,
    _tx.data
);
```

Discussion

rcstanciu

Comment from Optimism

Severity: medium

Reason: see 109

zobront

Escalate for 250 USDC

This report should not be a duplicate of #109.

For an issue to be judged as High, it requires the Watson to (a) understand the exploit and (b) provide a report with an adequate burden of proof for a High Severity Finding. This report does neither.

While the same underlying root cause is identified as #109, the Watson does not see the possibility that this could be used to brick legitimate withdrawals. Instead, it's seen merely as something that might happen accidentally. As a result, the Watson files the report as a Medium Severity, which is appropriate for what he found.

As a example, if you look at #109, you'll see:

- an explicit exploit vector that justified High Severity
- calculations for exactly how much extra gas is needed
- a full, runnable POC
- submitted as a high, accepted as a high

For this issue, we see:

- thinks it can only happen by accident
- no detailed explanation or gas calculation
- no POC
- submitted as only a medium
- optimism's comment is that it was accepted as just a medium

I understand that Optimism's team simply lumped them together because they address the same issue. But based on Sherlock's rules, issues of different severity should not be dup'd together. For that reason, we believe this should be dedup'd from #109 and live as its own issue.

sherlock-admin

> Escalate for 250 USDC
>
> This report should not be a duplicate of #109.

For an issue to be judged as High, it requires the Watson to (a) understand the exploit and (b) provide a report with an adequate burden of proof for a High Severity Finding. This report does neither.

While the same underlying root cause is identified as #109, the Watson does not see the possibility that this could be used to brick legitimate withdrawals. Instead, it's seen merely as something that might happen accidentally. As a result, the Watson files the report as a Medium Severity, which is appropriate for what he found.

As a example, if you look at #109, you'll see:

- an explicit exploit vector that justified High Severity
- calculations for exactly how much extra gas is needed
- a full, runnable POC
- submitted as a high, accepted as a high

For this issue, we see:

- thinks it can only happen by accident
- no detailed explanation or gas calculation
- no POC
- submitted as only a medium
- optimism's comment is that it was accepted as just a medium

I understand that Optimism's team simply lumped them together because they address the same issue. But based on Sherlock's rules, issues of different severity should not be dup'd together. For that reason, we believe this should be dedup'd from #109 and live as its own issue.

You've created a valid escalation for 250 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Evert0x

Escalation accepted.

Labeling as a unique medium (instead of duplicate high)

It's clear that the Watson didn't understand the full impact of the issue.

sherlock-admin

Escalation accepted.

Labeling as a unique medium (instead of duplicate high)

It's clear that the Watson didn't understand the full impact of the issue.

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

Issue M-9: Migration can be bricked by sending a message directly to the LegacyMessagePasser

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/105

Found by

0xdeadbeef, obront, Bobface, unforgiven, xiaoming90

Summary

The migration process halts and returns an error if any of the withdrawal data breaks from the specified format. However, the data for this migration comes from every call that has been made to the LegacyMessagePasser (0x00) address, and it is possible to send a transaction that would violate the requirements. The result is that the migration process would be bricked and need to be rebuilt, with some difficult technical challenges that we'll outline below.

Vulnerability Detail

Withdrawal data is saved in l2geth whenever a call is made to the LegacyMessagePasser address:

```
if addr == dump.MessagePasserAddress {
    statedumper.WriteMessage(caller.Address(), input)
}
```

This will save all the calls that came via the L2CrossDomainMessenger. The expected format for the data is encoded in the L2CrossDomainMessenger. It encodes the calldata to be executed on the L1 side as:
`abi.encodeWithSignature("relayMessage(...)", target, sender, message, nonce)`

The migration process expects the calldata to follow this format, and expects the call to come from L2CrossDomainMessenger, implemented with the following two checks:

```
selector :=
↪  crypto.Keccak256([]byte("relayMessage(address,address,bytes,uint256)"))[0:4]
if !bytes.Equal(data[0:4], selector) {
    return fmt.Errorf("invalid selector: 0x%x", data[0:4])
}

msgSender := data[len(data)-len(predeploys.L2CrossDomainMessengerAddr):]
if !bytes.Equal(msgSender, predeploys.L2CrossDomainMessengerAddr.Bytes()) {
```

```
    return errors.New("invalid msg.sender")
}
```

The migration process will be exited and the migration will fail if this assumption is violated.

However, since the function on the LegacyMessagePasser is public, it can also be called directly with arbitrary calldata:

```
function passMessageToL1(bytes memory _message) external {
    sentMessages[keccak256(abi.encodePacked(_message, msg.sender))] = true;
}
```

This allows us to submit calldata that would violate both of these checks and cause the migration to panic and fail.

While it may seem easy to filter these withdrawals out and rerun the migration, this solution would not work either. That's because, later in the process, we check that easy storage slot in the LegacyMessagePasser contract has a corresponding withdrawal in the migration:

```
for slot := range slotsAct {
    _, ok := slotsInp[slot]
    if !ok {
        return nil, fmt.Errorf("unknown storage slot in state: %s", slot)
    }
}
```

The result is that the Optimism team would need to unwind the migration, develop a new migration process to account for this issue, and remigrate with an untested system.

Impact

Exploitation of this bug would lead to significant challenges for the Optimism team, needing to run a less tested migration process (which could lead to further issues), and a significant amount of FUD in pausing a partially completed migration partway through. We think that the ability to unexpectedly shut down the migration causes enough structural damage as well as second-order financial damage to warrant high severity.

Code Snippet

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/l2geth/core/vm/evm.go#L207-L209

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/op-chain-ops/crossdomain/legacy_withdrawal.go#L58-L66

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts/contracts/L2/predeploys/OVM_L2ToL1MessagePasser.sol#L29-L34

Tool used

Manual Review

Recommendation

Rather than throwing an error if withdrawal data doesn't meet the requirements, save a list of these withdrawals and continue. Include this list when prechecking withdrawals to ensure that they are included in the storage slot matching process, but not included in withdrawals to be transferred to the new system.

Special note

After coming up with this attack, we've noticed that someone has done exactly what we described and sent a message directly to the MessagePasser! Obviously this TX has nothing to do with us and we want to make sure Optimism is absolutely safe during migration. Furthermore, this TX should be traced and if a contestant is linked to this then they should clearly be disqualified from being rewarded.

Issue M-10: Challenger can override the 7 day finalization period

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/75

Found by

unforgiven, csanuragjain

Summary

All withdrawals are finalized after a 7 days window (finalization period). After this duration transaction are confirmed and user can surely withdraw their balance. But due to lack of check, challenger can delete a l2Output which is older than 7 days meaning withdrawals will stop working for even confirmed transaction

Vulnerability Detail

1. Proposer has proposed L2 output for a _l2BlockNumber which creates entries on l2Outputs using the proposeL2Output. Assume this creates a new l2Output at index X

```
l2Outputs.push(
            Types.OutputProposal({
                outputRoot: _outputRoot,
                timestamp: uint128(block.timestamp),
                l2BlockNumber: uint128(_l2BlockNumber)
            })
        );
```

2. proveWithdrawalTransaction has been called for user linked to this l2Output
3. Finalization period(7 day) is over after proposal and Users is ready to call `finalizeWithdrawalTransaction` to withdraw their funds
4. Since confirmation is done, User A is sure that he will be able to withdraw and thinks to do it after coming back from his holidays
5. Challenger tries to delete the index X (Step 1), ideally it should not be allowed as already confirmed. But since there is no such timeline check so the l2Output gets deleted

```
function deleteL2Outputs(uint256 _l2OutputIndex) external {
        require(
            msg.sender == CHALLENGER,
            "L2OutputOracle: only the challenger address can delete outputs"
        );
```


SHERLOCK

```
        // Make sure we're not *increasing* the length of the array.
        require(
            _l2OutputIndex < l2Outputs.length,
            "L2OutputOracle: cannot delete outputs after the latest output index"
        );

        uint256 prevNextL2OutputIndex = nextOutputIndex();

        // Use assembly to delete the array elements because Solidity doesn't
↪   allow it.
        assembly {
            sstore(l2Outputs.slot, _l2OutputIndex)
        }

        emit OutputsDeleted(prevNextL2OutputIndex, _l2OutputIndex);
    }
```

6. User comes back and now tries to withdraw but the withdraw fails since the l2Output index X does not exist anymore. This is incorrect and nullifies the network guarantee.

Note: In case of a separate output root could be proven then user withdrawal will permanently stuck. Ideally if such anomaly could not be caught within finalization period then user should be allowed to withdraw

Impact

Withdrawal will fail for confirmed transaction

Code Snippet

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/L2OutputOracle.sol#L128-L148

Tool used

Manual Review

Recommendation

Add below check in

```
require(getL2Output(_l2OutputIndex).timestamp<=FINALIZATION_PERIOD_SECONDS,
↪   "Output already confirmed");
```

Discussion

rcstanciu

Comment from Optimism

Description: deleteL2Outputs delays finalization

Reason: This is know and expected behavior. If it did not delay finalization, new outputs would not be subject to the finalization window

csanuragjain

Escalate for 30 USDC

Finalization period is the time for raising any dispute and if finalization period is crossed then no matter what, transaction should be considered confirmed (Refer https://github.com/ethereum-optimism/optimism/blob/develop/specs/withdrawals.md).

The issue here is that Challenger is able to bypass the 7 days finalization period. Post 7 day finalization period, transaction should come in confirmed state but seems like Challenger can delete the l2OutputRoot even for confirmed state transaction. This is incorrect and will bring confirmed transaction into unconfirmed state again even when they completed the finalization period.

Challenger should be disallowed to delete any L2Output where `getL2Output(_l2OutputIndex).timestamp>FINALIZATION_PERIOD_SECONDS`

sherlock-admin

> Escalate for 30 USDC
>
> Finalization period is the time for raising any dispute and if finalization period is crossed then no matter what, transaction should be considered confirmed (Refer https://github.com/ethereum-optimism/optimism/blob/develop/specs/withdrawals.md).
>
> The issue here is that Challenger is able to bypass the 7 days finalization period. Post 7 day finalization period, transaction should come in confirmed state but seems like Challenger can delete the l2OutputRoot even for confirmed state transaction. This is incorrect and will bring confirmed transaction into unconfirmed state again even when they completed the finalization period.
>
> Challenger should be disallowed to delete any L2Output where `getL2Output(_l2OutputIndex).timestamp>FINALIZATION_PERIOD_SECONDS`

You've created a valid escalation for 30 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Evert0x

Escalation accepted.

Accepting escalation and labeling issue as medium as challenger is able to delete finalized withdrawals.

sherlock-admin

> Escalation accepted.
>
> Accepting escalation and labeling issue as medium as challenger is able to delete finalized withdrawals.

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

Issue M-11: Gasless ETH bridging from L1 to L2

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/71

Found by

tnch

Summary

Vulnerability Detail

Users can submit deposit transactions on L1 using the depositTransaction of the `OptimismPortal` contract. Among other parameters, the user must specify the gas limit.

The function does not enforce any minimum gas limit to pay for the submission of the transaction on L2. Therefore, it is possible to submit deposit gasless transactions (i.e., with a gas limit of zero). These gasless transactions trigger a number of interesting consequences on the system.

First, because the gas limit is set to zero, the deposit transaction won't accumulate any gas on the prevBoughtGas on L1. Therefore, they're not constrained by the MAX_RESOURCE_LIMIT.

Second, the resource and gas cost on L1 will be zero. And therefore they will never trigger the gas burning mechanism on L1. Users don't have to pay for the consumed L2 resources in advance.

Third, when the transaction is executed on L2, it will be run with a gas limit of zero. Unsurprisingly, execution will fail. This is a reasonable behavior. Although under certain circumstances, it will still allow the user to alter L2 state. In particular, if a user submits on L1 a deposit transaction with zero gas limit and positive `msg.value`, it will record a mint of the corresponding ETH on L2, even if the transaction fails. Thus effectively allowing users to bridge ETH from L1 to L2 in a gasless fashion, without paying in advance for the L2 resources consumed.

Fourth, it is worth noting that in `op-geth`, a comment states that "Failed deposits must still be included. Unless we cannot produce the block at all due to the gas limit". However, these gasless deposits bypass the block gas limit, and in this aspect guarantee inclusion. Because they consume no gas at all.

Here's a real example of gasless ETH bridging. In this L1 tx, the account `0x612c` calls `depositTransaction` with `msg.value` of 1 wei and `gasLimit` set to zero. The transaction executes successfuly. Some seconds later, the corresponding L2 tx is executed. It fails, but it increases account's ETH balance on L2 by 1 wei.

I should mention that the specs do explain the behavior of failed deposits, explicitly

SHERLOCK

stating that the mint on L2 will succeed and the nonce will be increased. However, it assumes the costs of the operation (even if failed) have already been paid on L1. This issue would prove the assumption incorrect.

Impact

Gasless bridging of ETH allows users to only pay for the L1 execution of their ETH deposit transactions. They allow users to spend L2-bridging resources without paying for them in advance on L1.

Also, the issue renders the expected way of bridging ETH pointless. Because the default for an ETH deposit to L2 uses a gas limit of 100000 units.

Code Snippet

Tool used

Manual review

Recommendation

Enforce on L1 a minimum gas limit that the users must pay for to successfully begin invocation on L2. I would suggest exploring the possibility to at least charge the intrinsic gas cost of the transaction on L1, so as to ensure the user is paying for the minimum gas a transaction must have to even be considered on L2.

Discussion

rcstanciu

Comment from Optimism

Description: Gasless ETH bridging from L1 to L2

Reason: Enables a dos and circumvents the resource metering logic.

0xunforgiven

Escalate for 51 USDC

shouldn't be Medium as the protocol representatives said in the walkthrough streams: "If you pay gas to consume resource in the L2, then it's not an issue" and here user need to pay L1 gas and it's not effective to DOS L2 by paying L1 gas which has high price. in any case even if user set 0 gas limit when calling deposits it can create 3500 deposits in one L1 block because calling portal consumes a lot of gas already. This can't be Dos because it would require a lot of L1 gas to perform DOS and adding 21K minimum gas doesn't change anything as user need to pay

more than 21K gas to call optimism portal. and it doesn't circumvents the resource metering logic because the function call uses 21K gas in L1.

sherlock-admin

> Escalate for 51 USDC
>
> shouldn't be Medium as the protocol representatives said in the walkthrough streams: "If you pay gas to consume resource in the L2, then it's not an issue" and here user need to pay L1 gas and it's not effective to DOS L2 by paying L1 gas which has high price. in any case even if user set 0 gas limit when calling deposits it can create 3500 deposits in one L1 block because calling portal consumes a lot of gas already. This can't be Dos because it would require a lot of L1 gas to perform DOS and adding 21K minimum gas doesn't change anything as user need to pay more than 21K gas to call optimism portal. and it doesn't circumvents the resource metering logic because the function call uses 21K gas in L1.

You've created a valid escalation for 51 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Evert0x

Escalation rejected.

It's a serious issue for Optimism, it could allow the creation of 4 MB deposit blocks. Although the attack is expensive for the attacker it is also costly to the sequencer to process such blocks without compensation.

sherlock-admin

> Escalation rejected.
>
> It's a serious issue for Optimism, it could allow the creation of 4 MB deposit blocks. Although the attack is expensive for the attacker it is also costly to the sequencer to process such blocks without compensation.

This issue's escalations have been rejected!

Watsons who escalated this issue will have their escalation amount deducted from their next payout.

Issue M-12: Deposits from L1 to L2 using L1CrossDomainMessenger will fail and will not be replayable when L2CrossDomainMessenger is paused

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/57

Found by

rvierdiiev, Bnke0x0, 0xdeadbeef

Summary

Deposits from L1 to L2 using L1CrossDomainMessenger will fail and will not be replayable when L2CrossDomainMessenger is paused.

Vulnerability Detail

Both `L1CrossDomainMessenger` and `L2CrossDomainMessenger` extend `CrossDomainMessenger` contract that can be paused.

When `CrossDomainMessenger` is paused then only `relayMessage` function will revert as it uses `whenNotPaused` modifier. It's still possible to call `sendMessage` when contract is paused.

In known issues section you can find following:

> 3.If the L1CrossDomainMessenger is paused, withdrawals sent to it will fail and not be replayable.

However there is another problem with pausing. In case if `L2CrossDomainMessenger` is paused and no matter if `L1CrossDomainMessenger` is pause or not(as `sendMessage` function is callable when contract is paused), then all deposits that are coming from L1 to L2 through the `L1CrossDomainMessenger` will fail on L2 side, as `relayMessage` will revert, because is paused. As result, it will be not possible for depositor to replay that deposit anymore as no information about the call is saved. All sent value will be minted to aliased `L2CrossDomainMessenger` address.

Example. 1.User wants to send himself some eth from L1 to L2 through `L1CrossDomainMessenger`. 2.When new `TransactionDeposited` event was fired by `OptimismPortal`, `L2CrossDomainMessenger` on L2 was paused for some reasons. 3.`TransactionDeposited` was explored by special service that sent tx to L2 node. 4.Sent value amount is minted to aliased `L2CrossDomainMessenger` address and `L2CrossDomainMessenger.relayMessage` is called which reverts. 5.Depositor lost his eth as he is not able to replay tx.

SHERLOCK

Impact

Lose of funds for depositor.

Code Snippet

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L212-L343

Tool used

Manual Review

Recommendation

Pause `CrossDomainMessenger` on both sides at same time(L1 and L2) and restrict call of `sendMessage` function when contract is paused.

Discussion

rcstanciu

Comment from Optimism

Description: lock ups on l1->l2 pausing

Issue M-13: Withdrawal transactions can get stuck if output root is reproposed

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/53

Found by

Allarious, Barichek, HE1M, cmichel, unforgiven

Summary

Withdrawal transactions may never be executed if the L2 output root for the block, for which the withdrawal was proven, is challenged and reproposed.

Vulnerability Detail

Withdrawal transactions can be reproven in the case that the output root for their previously proven output index has been updated. This can happen if the L2 output root was removed by the challenger. However, to circumvent malicious users from reproving messages all the time and resetting the withdrawal countdown, reproving can only be done on the same L2 block number (and if the output root changed).

If the challenger deletes the block with the withdrawal transaction and the proposer proposes a different block that does *not* have the withdrawal transaction, the withdrawal transaction can never be finalized - even if a future block includes the legitimate withdrawal transaction again, as reproving it is bound to the old `provenWithdrawals[withdrawalHash].l2OutputIndex`.

Impact

Legitimate withdrawal transactions will never be finalized if the proposed block was challenged and replaced with a different one not having the withdrawal transaction. As this call fails on the "lowest level", the `OptimismPortal`, these transactions also cannot be replayed or be issued refunds. In case the withdrawal transaction was a token bridge transfer, the tokens are stuck on the other chain and cannot be recovered by the user.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/3f4b3c328153a8aa03611158b6984d624b17c1d9/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L196-L197 https://github.com/ethereum-optimism/optimism/blob/3f4b3c328153a8aa03611158b6984d624b17c1d9/packages/contracts-bedrock/contracts/L1/L2OutputOracle.sol#L128

Tool used

Manual Review

Recommendation

Loosen the restriction of reproving: Allow reproving *under a new L2 output index* whenever the output root of the proven output index changes. This still balances the other concern of malicious users reproving transactions to reset the withdrawal countdown well as in the case where the output root changed, the withdrawal needs to be proved again anyway to be finalized.

```
- require(
-     provenWithdrawal.timestamp == 0 ||
-         (_l2OutputIndex == provenWithdrawal.l2OutputIndex &&
-             outputRoot != provenWithdrawal.outputRoot),
-     "OptimismPortal: withdrawal hash has already been proven"
- );
+ require(
+   provenWithdrawal.timestamp == 0 ||
+     L2_ORACLE.getL2Output(provenWithdrawal.l2OutputIndex) !=
 ↪  provenWithdrawal.outputRoot,
+   "OptimismPortal: withdrawal hash has already been proven"
+ );
```

Discussion

rcstanciu

Comment from Optimism

Description: cannot reprove if you would need to do so on a different output index

Reason: Yeah, this LGTM.

Allarious

Escalate for 60 USDC

While this issue is mostly discussed here as something that can happen among honest actors, this can be an attack vector from the `proposer` role to users that disrupts the **liveness** of the system. This attack can lock the funds of many transactions and I believe this should be labelled as high.

The Attack

A `proposer` can block many transactions by submitting a faulty output root to the oracle and proving the transaction before the actual transaction reaches the L2. This can cause the total lock of funds and there is no way to recover the funds after.

Justification of this issue being high

- The `proposer` is not a trusted role, and the `challenger` role can not stop this since `provenWithdrawals` are already written to L1. Furthermore, Faulty proofs should not be able to lock funds up and they should just make temporary disturbance in the system.
- Can block many incoming transactions at any time, proposer can wait as long as he wants to maximize the attack value.
- There is no longer a guarantee for the liveness of the system since there is no guarantee anymore that withdrawal transactions from L1 can be executed.
- This attack can be escalated if `proposer` and the `sequencer` are working together. Which means that user of the system who deposits into L2 has to trust Optimism not to lock up their funds (at the start of the project), which makes a centralized authority for L2.
- Since the `proposer` role is going to be decentralized in the future, this can have catastrophic outcomes.

The variations of this attack

- Proposer can target several transactions in L1 mempool or receive them from the sequencer on L2 to lock up. He can attack as many transaction before the `outputroot` is challenged.
- Proposer can submit an `outputroot` where all the leaves of the tree is set to true, in this case, everyone in the network will be able to front-run each other to lock up each others transactions. (Such output root can be built with O(log(n))
- If the proposer unknowingly sends an incorrect output root due to a faulty off-chain program, an attacker could exploit this by using it to obstruct all the incorrectly included withdrawal proofs, even if the proposer is not intentionally acting malicious.

sherlock-admin

> Escalate for 60 USDC
>
> While this issue is mostly discussed here as something that can happen among honest actors, this can be an attack vector from the `proposer` role to users that disrupts the **liveness** of the system. This attack can lock

the funds of many transactions and I believe this should be labelled as high.

The Attack

A `proposer` can block many transactions by submitting a faulty output root to the oracle and proving the transaction before the actual transaction reaches the L2. This can cause the total lock of funds and there is no way to recover the funds after.

Justification of this issue being high

- The `proposer` is not a trusted role, and the `challenger` role can not stop this since `provenWithdrawals` are already written to L1. Furthermore, Faulty proofs should not be able to lock funds up and they should just make temporary disturbance in the system.
- Can block many incoming transactions at any time, proposer can wait as long as he wants to maximize the attack value.
- There is no longer a guarantee for the liveness of the system since there is no guarantee anymore that withdrawal transactions from L1 can be executed.
- This attack can be escalated if `proposer` and the `sequencer` are working together. Which means that user of the system who deposits into L2 has to trust Optimism not to lock up their funds (at the start of the project), which makes a centralized authority for L2.
- Since the `proposer` role is going to be decentralized in the future, this can have catastrophic outcomes.

The variations of this attack

- Proposer can target several transactions in L1 mempool or receive them from the sequencer on L2 to lock up. He can attack as many transaction before the `outputroot` is challenged.
- Proposer can submit an `outputroot` where all the leaves of the tree is set to true, in this case, everyone in the network will be able to front-run each other to lock up each others transactions. (Such output root can be built with O(log(n))
- If the proposer unknowingly sends an incorrect output root due to a faulty off-chain program, an attacker could exploit this by using it to obstruct all the incorrectly included withdrawal proofs, even if the proposer is not intentionally acting malicious.

You've created a valid escalation for 60 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Evert0x

Escalation rejected.

Although the statements in the justifications section seem all correct to me, we take into account that the proposer role will be controlled by Optimism, this restriction makes the attack extremely unlikely as it requires a private key to be leaked.

Because Optimism documented plans to decentralize this role we keep it medium severity.

sherlock-admin

> Escalation rejected.
>
> Although the statements in the justifications section seem all correct to me, we take into account that the proposer role will be controlled by Optimism, this restriction makes the attack extremely unlikely as it requires a private key to be leaked.
>
> Because Optimism documented plans to decentralize this role we keep it medium severity.

This issue's escalations have been rejected!

Watsons who escalated this issue will have their escalation amount deducted from their next payout.

Issue S-1: Batch validation logic is ordered differently to specification

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/283

Found by

obront

Summary

Optimism L2 block derivation has a long list of validation steps performed on incoming blocks. They are described in the spec page, but the code implements the validations in a different order than the spec lays out.

Vulnerability Detail

There is an error in the order which is described in the spec versus the one in the implementation.

According to the specs, the following check comes first:

> `batch.timestamp > batch_origin.time + max_sequencer_drift -> drop: i.e. a batch that does not adopt the next L1 within time will be dropped, in favor of an empty batch that can advance the L1 origin. This enforces the max L2 timestamp rule.`

After it comes this check:

> `batch.timestamp < batch_origin.time -> drop: enforce the min L2 timestamp rule.`

However, in code in `batches.go` executes in reverse:

```
if batch.Batch.Timestamp < batchOrigin.Time {
    log.Warn("batch timestamp is less than L1 origin timestamp", "l2_timestamp",
    ↪  batch.Batch.Timestamp, "l1_timestamp", batchOrigin.Time, "origin",
    ↪  batchOrigin.ID())
    return BatchDrop
}
// If we ran out of sequencer time drift, then we drop the batch and produce an
    ↪  empty batch instead,
// as the sequencer is not allowed to include anything past this point without
    ↪  moving to the next epoch.
if max := batchOrigin.Time + cfg.MaxSequencerDrift; batch.Batch.Timestamp > max {
    log.Warn("batch exceeded sequencer time drift, sequencer must adopt new L1
    ↪  origin to include transactions again", "max_time", max)
```


SHERLOCK

```
    return BatchDrop
}
```

In practice this should not lead to any issues because both are drop rules and are right next to each other.

Impact

The code does not align with the ordering laid out in the specification.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/407f97b9d13448b766624995ec824d3059d4d4f6/op-node/rollup/derive/batches.go#L98

Tool used

Manual Review

Recommendation

Change the code so that it is in line with the specification.

SHERLOCK

Issue S-2: Specification of the new base fee calculation is inconsistent with the code

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/266

Found by

shw

Summary

Specification of the new base fee calculation is inconsistent with the code.

Vulnerability Detail

The "Guaranteed Gas Fee Market" specification provides a pseudocode explaining how a new base fee is calculated, which has two inconsistencies with the actual code implementation:

1. When multiple blocks are skipped, the new base fee is applied with a 7/8 reduction per block. The code uses the updated `newBaseFee` (see `ResourceMetering.sol#L115`) while the specification uses the un-updated `prev_basefee`.
2. The maximum new base fee in the specification is `UINT_64_MAX_VALUE` but is set to `type(uint128).max` in the code.

Impact

The specification does not match the code.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/3f4b3c328153a8aa03611158b6984d624b17c1d9/packages/contracts-bedrock/contracts/L1/ResourceMetering.sol#L115

Tool used

Manual Review

Recommendation

Fix either the specification or the code.

SHERLOCK

Issue S-3: Incorrect code comments in the `StandardBridge.sol` contract

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/264

Found by

unforgiven, tsvetanovv, shw

Summary

Incorrect code comments in the `StandardBridge.sol` contract.

Vulnerability Detail

The code comments for the `bridgeERC20` and `bridgeERC20To` functions in the `universal/StandardBridge.sol` contract do not match the actual code. The comments say the bridge returns tokens to the sender if the bridging fails, but the refund logic has been removed since PR#3535.

Impact

Specfication error only.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/3f4b3c328153a8aa03611158b6984d624b17c1d9/packages/contracts-bedrock/contracts/universal/StandardBridge.sol#L219-L222 https://github.com/ethereum-optimism/optimism/blob/3f4b3c328153a8aa03611158b6984d624b17c1d9/packages/contracts-bedrock/contracts/universal/StandardBridge.sol#L251-L254

Tool used

Manual Review

Recommendation

Remove the outdated code comments.

SHERLOCK

Issue S-4: Incorrect owner check

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/263

Found by

csanuragjain

Summary

The depositTransaction of OptimismPortal can directly be called by user instead of intermediate contract. This means from address wont be aliased. But this is not considered in CrossDomainOwnable contract which plainly undoL1ToL2Alias the caller

Vulnerability Detail

1. Assume `depositTransaction` is called by User A directly. Since no intermediary contract so no aliasing is done
2. On L2 side, if _checkOwner is checked

```
function _checkOwner() internal view override {
        require(
            owner() == AddressAliasHelper.undoL1ToL2Alias(msg.sender),
            "CrossDomainOwnable: caller is not the owner"
        );
    }
```

3. This will try undoL1ToL2Alias on User A address and then match with owner which is incorrect since User A address was never aliased on L1

Impact

The owner check might fail for genuine transaction

Code Snippet

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L2/CrossDomainOwnable.sol#L21

Tool used

Manual Review

Recommendation

This check need to be revised. If the transaction came directly from tx.origin (without any intermediary contract) then no need of removing aliasing

Discussion

csanuragjain

Escalate for 28 USDC

This is mentioning a valid issue where CrossDomainOwnable.sol#L21 is always assuming call to `depositTransaction` coming from contract (as undo Alias is always done). If `depositTransaction` is called by a user instead then `CrossDomainOwnable` will fail to check properly and not allow a genuine user

sherlock-admin

> Escalate for 28 USDC
>
> This is mentioning a valid issue where CrossDomainOwnable.sol#L21 is always assuming call to `depositTransaction` coming from contract (as undo Alias is always done). If `depositTransaction` is called by a user instead then `CrossDomainOwnable` will fail to check properly and not allow a genuine user

You've created a valid escalation for 28 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Evert0x

Escalation accepted and labeling as specification issue

sherlock-admin

> Escalation accepted and labeling as specification issue

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

Issue S-5: Block number is not an input to `proveWithdrawalTransaction` in `OptimismPortal`

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/233

Found by

ck

Summary

According to the `Withdrawals` spec, a block number is one of the submitted inputs to the `OptimismPortal` which is not the case.

Vulnerability Detail

The documentation says:

"A relayer submits the required inputs to the OptimismPortal contract. The relayer need not be the same entity which initiated the withdrawal on L2. These inputs include the withdrawal transaction data, inclusion proofs, and a block number. The block number must be one for which an L2 output root exists, which commits to the withdrawal as registered on L2."

In the contract, the `_l2OutputIndex` is the one used to check for existence of the `outputRoot` and not the block number.

```
function proveWithdrawalTransaction(
    Types.WithdrawalTransaction memory _tx,
    uint256 _l2OutputIndex,
    Types.OutputRootProof calldata _outputRootProof,
    bytes[] calldata _withdrawalProof
) external {
    // Prevent users from creating a deposit transaction where this address is
↪  the message
    // sender on L2. Because this is checked here, we do not need to check
↪  again in
    // `finalizeWithdrawalTransaction`.
    require(
        _tx.target != address(this),
        "OptimismPortal: you cannot send messages to the portal contract"
    );

    // Get the output root and load onto the stack to prevent multiple mloads.
↪  This will
    // revert if there is no output root for the given block number.
    bytes32 outputRoot = L2_ORACLE.getL2Output(_l2OutputIndex).outputRoot;
```


SHERLOCK

```
    // Verify that the output root can be generated with the elements in the
↪  proof.
    require(
        outputRoot == Hashing.hashOutputRootProof(_outputRootProof),
        "OptimismPortal: invalid output root proof"
    );
```

Impact

Misleading spec

Code Snippet

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L160-L182

Tool used

Manual Review

Recommendation

The specification should refer to `_l2OutputIndex` as one of the inputs instead of 'block number'.

Issue S-6: `system_config`: incorrect variable name and missing config update type

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/213

Found by

lemonmon

Summary

The specs for system config contains incorrect information and incorrect names for its contents.

1) The names for `overhead` and `scalar` are falsely stated as `l1FeeOverhead` and `l1FeeScalar` in multiple occasions.
2) The list of the configuration update types is missing the **unsafe block signer update**

Vulnerability Detail

In the `SystemConfig` contract, there are public variables `overhead` and `scalar`:

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/SystemConfig.sol#L51-L59

Below the variables `overhead` and `scalar` are incorrectly named as `l1FeeOverhead` and `l1FeeScalar` in multiple occasions:

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/system_config.md?plain=1#L34 https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/system_config.md?plain=1#L73 https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/system_config.md?plain=1#L86-L90

In the above snippet: Line 89: the names of `overhead` and `scalar` It also is missing the type 3:

- type `3`: `unsafeBlockSigner` overwrite, as `address` payload.

The corresponding update code snippets from `SystemConfig` are below:

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/SystemConfig.sol#L25-L30 https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/SystemConfig.sol#L169-L175

Impact

factually incorrect information

- incorrect name of variables
- missing config update type

As the name of variables double as interface to fetch the value, anybody uses the incorrect name in the specs to fetch the values will fail. Also the specs do not list the possible config update types, so the users may not know that the `unsafeBlockSigner` can be updated.

Code Snippet

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/SystemConfig.sol#L51-L59 https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/system_config.md?plain=1#L34 https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/system_config.md?plain=1#L73 https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/system_config.md?plain=1#L86-L90 https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/SystemConfig.sol#L25-L30 https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/SystemConfig.sol#L169-L175

Tool used

Manual Review

Recommendation

Correct the names of the variables and add the missing config update type

Issue S-7: Address collision in cross - chain contract creation (breaks tooling)

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/204

Found by

Bahurum, Ro

Summary

When creating a contract from L1 (mainnet) to L2 (optimism) the nonce of the sender is kept at 0 in the context of the transaction receipt and thus the existing infrastructure (Etherscan) will incorrectly interpret the address of the new contract (creating a collision).

Vulnerability Detail

It is possible to create a contract in Layer 2 from Layer 1 by sending a transaction to the optimism portal. This is the function of interest:

```
function depositTransaction(
       address _to,
       uint256 _value,
       uint64 _gasLimit,
       bool _isCreation,
       bytes memory _data
   ) public payable metered(_gasLimit)
```

To create a contract the "_to" address needs to be set to "address(0x0), " the "_isCreation" needs to be set to "true" and the bytecode needs to be passed in the "_data" field.

- The contract is then created through the "create" opcode, therefore the address is computed by hashing the nonce and the address of the sender.

If the nonce is kept at 0 during some time of the execution, the new contract address will always be the same (nonce is always 0).

NOTE: In reality, the new contract is created with the proper nonce (and the nonce is updated), but the receipt displays it wrongly and during a point in the execution is also kept at 0 so it breaks Etherscan, Blockscout and the rest of the block exploreres.

Impact

Infrastructure tooling like Etherscan is the source of truth for most users, therefore having inconsistent data between these tools and Optimism is not recommended.

This bug causes existing tooling like Etherscan and Blockscout to display wrong data in the following ways:

1. The first contract created will always appear as the new contract in contract creation. See this link for reference: https://goerli-optimism.etherscan.io/address/0x5d7ee88447367b9212fa655dc863a1e83f7e189d

When the account 0x7d.. creates a new contract from Layer 1, the address "0x5d.." will always appear as the new contract created.

2. It incorrectly displays contracts as EOA's See this address on etherscan goerli optimism: https://goerli-optimism.etherscan.io/address/0x0a1c3c13c35275d030ff9fb660946cf2fca74ece It appears to be a regular EOA, while in reality, it is a contract. Run the following command to verify:

```
cast code 0x0a1c3c13c35275d030ff9fb660946cf2fca74ece --rpc-url
↪  https://goerli.optimism.io
```

Code Snippet

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L358

Tool used

Manual Review

Recommendation

The nonce should be set to the sender's nonce from the op-node.

Discussion

rcstanciu

Comment from Optimism

Description: Address collision in cross - chain contract creation (breaks tooling)

Reason: This is correct, but is unexpected behavior external to the scope of the code. Thus, this is a low issue.

Action: Increment the nonce in the op-node to make transaction receipts use the correct nonce for contract creation on L1 -> L2 bridge.

rodrigoherrerai

Escalate for 250 USDC

This bug clearly breaks the specifications from Optimism. The debate here is if this is marked as a Medium or as a spec.

It could be marked as medium because it breaks EVM complete equivalence (as shown below):

> Bedrock's design aims for EVM equivalence, meaning that smart contract developers should be able to safely deploy the same contracts to Optimism as they would on Ethereum, without concern for differences in the execution.

Here are concrete examples where this bug causes the specifications to differ:

Just to recap, this issue is manifested when creating contracts from L1 -> L2. The nonce is not incremented and thus the contract created in the transaction receipt will always be the same (nonce is always 0).

When creating a contract, the address is computed as follows:

```
computed_address = keccak256(rlp.encode([address, nonce]))
    canonical_address = computed_address[-20:]
    padded_address = left_pad_zero_bytes(canonical_address, 20)
    return Address(padded_address)
```

If the nonce is kept at 0 during the receipt, then the new contract address will always be the same. https://github.com/ethereum/execution-specs/blob/master/src/ethereum/london/utils/address.py#L42

This issue breaks the following specifications:

1:

> **Nonce Handling** Despite the lack of signature validation, we still increment the nonce of the from account when a deposit transaction is executed. In the context of a deposit-only roll up, this is not necessary for transaction ordering or replay prevention, however it maintains consistency with the use of nonces during contract creation. It may also simplify integration with downstream tooling (such as wallets and block explorers).

Link: https://github.com/ethereum-optimism/optimism/blob/develop/specs/deposits.md#nonce-handling

Quote: "however it maintains consistency with the use of nonces during

contract creation. It may also simplify integration with downstream tooling (such as wallets and block explorers)."

This issue completely breaks the integration with downstream tooling:

- Etherscan and the rest of the block explorers show contract accounts without code (as EOA's):

This is an account with code (contract) on Optimism goerli: 0x0a1c3c13c35275d030ff9fb660946cf2fca74ece

But Etherscan and the rest of the block explorers display it as an EOA:

- Etherscan and the rest of the block explorers incorrectly show the contract creation as the first contract will always appear to be the contract created:

Take this transaction for example:

It appears as if the contract created was "0x5d7..", while in reality it is incorrect. Note that the nonce also appears as "0".

2.

It breaks complete "EVM Equivalence"

From the Optimism blog post:

> ...What is EVM equivalence? In short: EVM equivalence is complete compliance with theEthereum yellow paper, the formal definition of the protocol. By definition, L1 Ethereum software must comply with this specification.
>
> This means that — down to the very deepest depths — the existing Ethereum stack will now integrate with the L2 system, too. Every debugger. Every toolchain. Every node implementation. We believe that any L2 offering any EVM experience will have to meet this bar — anything less is unacceptable.

Quote: "This means that — down to the very deepest depths — the existing Ethereum stack will now integrate with the L2 system, too. Every debugger. Every toolchain. Every node implementation. We believe that any L2 offering any EVM experience will have to meet this bar — anything less is unacceptable." Link: https://medium.com/ethereum-optimism/introducing-evm-equivalence-5c2021deb306

Apart from breaking downstream tooling like block explorers and debuggers, it can also break frontend / backend applications that rely on the transaction receipt as the source of truth.

For example, when the address 0x7dF608C479d2D2a7df677fEEc6f2535Eb268da01 creates a contract from L1 to L2, the contract address in the receipt will always be the same, because the nonce is kept at 0:

```
{
  to: null,
  from: '0x7dF608C479d2D2a7df677fEEc6f2535Eb268da01',
  contractAddress: '0x5d7Ee88447367b9212FA655dc863A1e83f7e189D',
...
```

sherlock-admin

> Escalate for 250 USDC
>
> This bug clearly breaks the specifications from Optimism. The debate here is if this is marked as a Medium or as a spec.
>
> It could be marked as medium because it breaks EVM complete equivalence (as shown below):
>
> > Bedrock's design aims for EVM equivalence, meaning that smart contract developers should be able to safely deploy the same contracts to Optimism as they would on Ethereum, without concern for differences in the execution.
>
> Here are concrete examples where this bug causes the specifications to differ:
>
> Just to recap, this issue is manifested when creating contracts from L1 -> L2. The nonce is not incremented and thus the contract created in the transaction receipt will always be the same (nonce is always 0).
>
> When creating a contract, the address is computed as follows:

```
computed_address = keccak256(rlp.encode([address, nonce]))
    canonical_address = computed_address[-20:]
    padded_address = left_pad_zero_bytes(canonical_address, 20)
    return Address(padded_address)
```

> If the nonce is kept at 0 during the receipt, then the new contract address will always be the same. https://github.com/ethereum/execution-specs/blob/master/src/ethereum/london/utils/address.py#L42
>
> This issue breaks the following specifications:
>
> 1:
>
> > **Nonce Handling** Despite the lack of signature validation, we still increment the nonce of the from account when a deposit transaction is executed. In the context of a deposit-only roll up, this is not necessary for transaction ordering or replay prevention, however it maintains consistency with the use of

> nonces during contract creation. It may also simplify integration with downstream tooling (such as wallets and block explorers).

Link: https://github.com/ethereum-optimism/optimism/blob/develop/specs/deposits.md#nonce-handling

Quote: "however it maintains consistency with the use of nonces during contract creation. It may also simplify integration with downstream tooling (such as wallets and block explorers)."

This issue completely breaks the integration with downstream tooling:

- Etherscan and the rest of the block explorers show contract accounts without code (as EOA's):

This is an account with code (contract) on Optimism goerli: 0x0a1c3c13c35275d030ff9fb660946cf2fca74ece

But Etherscan and the rest of the block explorers display it as an EOA:

- Etherscan and the rest of the block explorers incorrectly show the contract creation as the first contract will always appear to be the contract created:

Take this transaction for example:

It appears as if the contract created was "0x5d7..", while in reality it is incorrect. Note that the nonce also appears as "0".

2.

It breaks complete "EVM Equivalence"

From the Optimism blog post:

> . . . What is EVM equivalence? In short: EVM equivalence is complete compliance with theEthereum yellow paper, the formal definition of the protocol. By definition, L1 Ethereum software must comply with this specification.
>
> This means that — down to the very deepest depths — the existing Ethereum stack will now integrate with the L2 system, too. Every debugger. Every toolchain. Every node implementation. We believe that any L2 offering any EVM experience will have to meet this bar — anything less is unacceptable.

Quote: "This means that — down to the very deepest depths — the existing Ethereum stack will now integrate with the L2 system, too. Every debugger. Every toolchain. Every node implementation. We believe that any L2 offering any EVM experience will have to meet this bar —

> anything less is unacceptable." Link: https://medium.com/ethereum-optimism/introducing-evm-equivalence-5c2021deb306
>
> Apart from breaking downstream tooling like block explorers and debuggers, it can also break frontend / backend applications that rely on the transaction receipt as the source of truth.
>
> For example, when the address 0x7dF608C479d2D2a7df677fEEc6f2535Eb268da01 creates a contract from L1 to L2, the contract address in the receipt will always be the same, because the nonce is kept at 0:

```
{
  to: null,
  from: '0x7dF608C479d2D2a7df677fEEc6f2535Eb268da01',
  contractAddress: '0x5d7Ee88447367b9212FA655dc863A1e83f7e189D',
...
```

You've created a valid escalation for 250 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Evert0x

Escalation accepted.

Comment provides sufficient evidence for a specification issue.

Labeling #176 as duplicate as it describes the same issue

sherlock-admin

> Escalation accepted.
>
> Comment provides sufficient evidence for a specification issue.
>
> Labeling #176 as duplicate as it describes the same issue

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

Issue S-8: Specification Inconsistency in Legacy ERC20 Functions Reverting

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/200

Found by

Robert

Summary

Documentation says that after Bedrock migration all methods interacting with state on the LegacyERC20 contract will now revert https://github.com/ethereum-optimism/optimism/blob/develop/specs/predeploys.md#legacyerc20eth. Functions that changed state were already reverting and view functions are being updated to still work.

Vulnerability Detail

Just not working as described.

Impact

Low

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/407f97b9d13448b766624995ec824d3059d4d4f6/packages/contracts-bedrock/contracts/legacy/LegacyERC20ETH.sol#L35

Tool used

Manual Review

Recommendation

Remove comment about this reverting--maybe change to it saying they're being updated (such as balance using address.balance).

Issue S-9: deposits: the `sourceHash` of L1 attributes deposited

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/162

Found by

Bahurum, lemonmon

Summary

The calculation of `sourceHash` for L1 attributes deposited is incorrect.

Although, it is a very small difference, because of the misplaced blacket, it means a different thing with a different result from the actual calculation.

Vulnerability Detail

According to the specs, the `sourceHash` of L1 attributes deposited is calculated based on:

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/deposits.md?plain=1#L92

It means the `l1BlockHash` will be hashed alone, before it is hashed with other values. However, `l1BlockHash` and `seqNumber` should be hashed together, as the actual calculation in the `deposit_source.go`:

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/op-node/rollup/derive/deposit_source.go#L35-L46

Therefore, the line should be corrected as following:

```
-  `keccak256(bytes32(uint256(1)), keccak256(l1BlockHash),
 ↪  bytes32(uint256(seqNumber)))`.
+  `keccak256(bytes32(uint256(1)), keccak256(l1BlockHash,
 ↪  bytes32(uint256(seqNumber))))`.
```

Impact

factually incorrect information

The calculation of `sourceHash` in the specs will give a different result from the actual code.

SHERLOCK

Code Snippet

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/deposits.md?plain=1#L92 https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/op-node/rollup/derive/deposit_source.go#L35-L46

Tool used

Manual Review

Recommendation

correct the calculation

Issue S-10: overview, withdrawals: DepositFeed does not exist

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/161

Found by

ck, Ro, lemonmon

Summary

There is no `DepositFeed` contract. The implementation of Deposit contract would be `OptimismPortal`. There are some multiple occasions of incorrect information, for example, stating that Optimism Portal inherits from `DepositFeed` contract.

Also, using "Deposit Contract" and `DepositFeed` contract interchangeably may confuse the reader.

Vulnerability Detail

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/overview.md?plain=1#L52

There is no `DepositFeed` contract. It should be `OptimismPortal` contract.

```
- The `OptimismPortal` contract emits `TransactionDeposited` events, which the
 ↪  rollup driver reads in order to process
```

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/overview.md?plain=1#L109

Here as well, the `DepositFeed` contract should be `OptimismPortal` contract.

```
call the `depositTransaction` method on the `OptimismPortal` contract. This in
 ↪  turn emits `TransactionDeposited` events,
```

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/overview.md?plain=1#L144

Here as well, the `DepositFeed` contract should be `OptimismPortal` contract.

```
deposits initiated via the `OptimismPortal` contract on L1. All L2 blocks can
 ↪  also contain _sequenced transactions_, i.e.
```

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/withdrawals.md?plain=1#L133-L135

SHERLOCK

The `OptimismPortal` inherits `Initializable`, `ResourceMetering` and `Semver` and there is no `DepositFeed` contract in the inheritance tree.

Impact

Factually wrong specs

Code Snippet

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/overview.md?plain=1#L52 https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/overview.md?plain=1#L109 https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/overview.md?plain=1#L144 https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/withdrawals.md?plain=1#L133-L135

Tool used

Manual Review

Recommendation

Use "Deposit Contract" or `OptimismPortal` depending on the context, instead of `DepositFeed` contract.

SHERLOCK

Issue S-11: Confusion in gap size

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/146

Found by

ustas, 0x1337, supernova

Summary

According to the comments

```
Reserve extra slots in the storage layout for future upgrades.
     *         A gap size of 41 was chosen here, so that the first slot used in
↪  a child contract
     *         would be a multiple of 50.
```

Vulnerability Detail

But actually gap is provided for 42 instead of 41 mentioned above . This can lead to presumptions on the minds of the dev that the first slot of the child contract is a multiple of 50 , when it is not .

Impact

Code Snippet

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L132-L137

Tool used

Manual Review

Recommendation

Clear the collision in the comments and the actual code .

SHERLOCK

Issue S-12: The CrossDomainMessenger process is explained incorrectly in the spec

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/108

Found by

obront, 0xdeadbeef, lemonmon

Summary

The process for using the L1CrossDomainMessenger has not been updated in the spec, and it still explains the pre-Bedrock process rather than the updated process.

Vulnerability Detail

The spec explains:

> When going from L2 into L1, the user must call relayMessage on the L1CrossDomainMessenger to finalize the withdrawal. This function can only be called after the finalization window has passed.

This is no longer the process. In Bedrock:

- the user proves their withdrawal right away
- the user proves their withdrawal on OptimismPortal, not CrossDomainMessenger
- the user executes their withdrawal on OptimismPortal, not by calling `relayMessage` (or any other function) on CrossDomainMessenger

Impact

The spec is still showing the old withdrawal process, and doesn't accurately reflect the new process that will exist in Bedrock.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/develop/specs/messengers.md#message-passing

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L160-L344

Tool used

Manual Review

Recommendation

Update the explanation of the CrossDomainMessenger in the spec to explain the new withdrawal process.

Issue S-13: Withdrawing process in spec does not include two-step withdrawals

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/107

Found by

rvierdiiev, obront, lemonmon

Summary

The withdrawal process in the Introduction of the spec displays the old, single-step withdrawal process, not the new, two-step one.

Vulnerability Detail

The Introduction section of the spec explains withdrawals with the following image:

https://github.com/ethereum-optimism/optimism/blob/develop/specs/assets/user-withdrawing-to-l1.svg

This image seems to have been drawn for the old withdrawal system, including steps:

```
4. Wait for block hash to finalize
5. Send execute withdrawal transaction
```

In the new system, the user submits the proof right away, then waits at least 7 days for the proof (regardless of when block hash finalizes), and then makes an additional call to execute the transaction.

Impact

The spec is still showing the old withdrawal process, and doesn't accurately reflect the new process that will exist in Bedrock.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/develop/specs/introduction.md#withdrawing

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L160-L344

Tool used

Manual Review

Recommendation

Update the diagram in the spec Introduction to explain the new two-step withdrawal process.

Issue S-14: Deposits are not guaranteed to be reflected within sequencing window

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/106

Found by

obront

Summary

There is an inconsistency between the spec and the code regarding guarantees for deposits.

Vulnerability Detail

In the spec, it states:

> Deposits are guaranteed to be reflected in the L2 state within the sequencing window.

However, until fraud proofs are implemented, there are no guarantees that this will actually be the case.

Impact

Users may expect that there are guarantees in the system that ensure their deposits will be processed within a given number of blocks, but these guarantees do not exist yet.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/develop/specs/overview.md#l1-components

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/L2OutputOracle.sol#L160-L210

Tool used

Manual Review

Recommendation

Remove this language from the spec until fraud proofs are live.

Discussion

rcstanciu

Comment from Optimism

Description: Deposits are not guaranteed without fault proofs

Reason: This is actually guaranteed, and part of the protocol definition for batch derivation.

zobront

Escalate for 250 USDC

This issue is a valid Low severity, and should be included.

Optimism's response was:

> Reason: This is actually guaranteed, and part of the protocol definition for batch derivation.

However, that isn't the case. In my conversation with Maurelian over DMs during the contest, he explained: "basically, you could say that the protocol guarantees this, based on the execution rules in the sequencer. However that 'guarantee' is based on the assumption that we have a fault proof working, which would enforce those execution rules."

Since we don't have a fault proof working, we do not have a guarantee.

sherlock-admin

> Escalate for 250 USDC
>
> This issue is a valid Low severity, and should be included.
>
> Optimism's response was:
>
> > Reason: This is actually guaranteed, and part of the protocol definition for batch derivation.
>
> However, that isn't the case. In my conversation with Maurelian over DMs during the contest, he explained: "basically, you could say that the protocol guarantees this, based on the execution rules in the sequencer. However that 'guarantee' is based on the assumption that we have a fault proof working, which would enforce those execution rules."
>
> Since we don't have a fault proof working, we do not have a guarantee.

You've created a valid escalation for 250 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Evert0x

Escalation accepted.

The specification is correct as it related the L2 state (not the L1 representation for the L2 state) but the context of the specification can be improved.

sherlock-admin

> Escalation accepted.
>
> The specification is correct as it related the L2 state (not the L1 representation for the L2 state) but the context of the specification can be improved.

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

Issue S-15: Information passed to L1Block.sol is not delayed by 10 blocks

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/103

Found by

obront

Summary

There is an inconsistency between the spec and the code regarding the timing of L1 block info being submitted to the L2 contract L1Block.sol.

Vulnerability Detail

In the spec, it states:

> Currently the L1 information is delayed by ten block confirmations (~2.5 minutes) to minimize the impact of reorgs. This value may be reduced in the future.

However, in the node's derivation code, the L1 block information is included along with the deposits for each block as it's processed, with no delays.

This can further be verified by watching L1 as well as the L1Block contract, and observing that, for a given block, the information is posted to Optimism instantly.

- https://goerli.etherscan.io/
- https://goerli-optimism.etherscan.io/address/0x4200000000000000000000000000000000000015

Impact

Spec doesn't accurately reflect the reality of what the code is doing.

Code Snippet

https://community.optimism.io/docs/developers/bedrock/differences/#l1block

https://github.com/ethereum-optimism/optimism/blob/6c6d142d7bb95faa11066aab5d8aed7187abfe38/op-node/rollup/derive/attributes.go#L56-L68

Tool used

Manual Review

Recommendation

Remove this language from the spec or adjust the code to match.

Issue S-16: L2OutputOracle outputs are removed by challenger, not proposer

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/98

Found by

tnch, obront, lemonmon

Summary

There is an inconsistency between the spec and reality regarding which user is able to delete the L2 outputs to roll back the chain.

Vulnerability Detail

In the spec, it states:

> The proposer may also delete multiple output roots by calling the deleteL2Outputs() function and specifying the index of the first output to delete, this will also delete all subsequent outputs.

It goes on to explicitly state that this will be the same role as the sequencer:

> Note regarding future work: In the initial version of the system, the proposer will be the same entity as the sequencer, which is a trusted role. In the future proposers will need to submit a bond in order to post L2 output roots, and some or all of this bond may be taken in the event of a faulty proposal.

However, the code specifies a different user, called `CHALLENGER`, who has this permission:

```
require(
    msg.sender == CHALLENGER,
    "L2OutputOracle: only the challenger address can delete outputs"
);
```

Looking at the `CHALLENGER` address, it appears that it is actually a multisig, separate from both the sequencer and proposer.

Impact

The spec is incorrect in defining which user has the ability to roll back the chain.

SHERLOCK

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/f30376825c82f62b846590487fe46b7435213d37/specs/proposals.md#proposing-l2-output-commitments

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/L2OutputOracle.sol#L129-L132

Tool used

Manual Review

Recommendation

Adjust the language in the spec to make clear that the `CHALLENGER` is a separate role controlled by a multisig.

Issue S-17: Spec: Wrong description of withdrawal process

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/94

Found by

ck, 0xdeadbeef, lemonmon, cmichel, obront, unforgiven

Summary

Wrong description of the withdrawal process in the spec.

Vulnerability Detail

The withdrawals.md describes the second step of the withdrawal process as:

> 2. The `OptimismPortal` contract retrieves the output root for the given block number from the `L2OutputOracle`'s `getL2OutputAfter()` function, and performs the remainder of the verification process internally.

The `getL2OutputAfter` call is never performed. This function does not even exist, most likely, it was referring to `getL2OutputIndexAfter`. However, even this function is not called. What happens in the withdrawal process is that `getL2Output` is called to retrieve the output root for the given block number (index).

Impact

The withdrawal process should be clearly documented in the spec. Currently, it's referring to a non-existant function.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/3f4b3c328153a8aa03611158b6984d624b17c1d9/specs/withdrawals.md?plain=1#L67

Tool used

Manual Review

Recommendation

Fix the spec by referring to `getL2Output(l2OutputIndex)` instead.

SHERLOCK

Issue S-18: Spec: Wrong `OptimismPortal` interface

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/93

Found by

cmichel, lemonmon

Summary

Wrong `OptimismPortal` interface in the specs.

Vulnerability Detail

The withdrawals.md specification file shows a wrong `L2ToL1MessagePasser` interface:

- `function proveWithdrawalTransaction(Types.WithdrawalTransaction memory _tx, uint256 _l2BlockNumber, Types.OutputRootProof calldata _outputRootProof, bytes[] calldata _withdrawalProof) external` uses the wrong `_l2BlockNumber` parameter. The parameter should be named `_l2OutputIndex` like in the `OptimismPortal` code. The difference is that the `L2OutputOracle` pushes the blocks to an array, and the `startingBlockNumber` is its first element, so there's a shift from a L2 output *index* to its L2 *block number*. Using the block number as described in the spec will make any `proveWithdrawalTransaction` calls fail.

```
interface OptimismPortal {

    event WithdrawalFinalized(bytes32 indexed);

    function l2Sender() returns(address) external;

    function proveWithdrawalTransaction(
        Types.WithdrawalTransaction memory _tx,
        uint256 _l2BlockNumber, // @audit this is NOT the block number, it's the
↪   `_l2OutputIndex`.
        Types.OutputRootProof calldata _outputRootProof,
        bytes[] calldata _withdrawalProof
    ) external;

    function finalizeWithdrawalTransaction(
        Types.WithdrawalTransaction memory _tx
    ) external;
}
```


SHERLOCK

Impact

Users usually go to the docs & specification to see how to integrate a project. Integrating Optimisim's `OptimismPortal` based on the specification will lead to errors as it uses the block number, different from the required block index in `L2OutputOracle`'s array.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/3f4b3c328153a8aa03611158b6984d624b17c1d9/specs/withdrawals.md?plain=1#L149

Tool used

Manual Review

Recommendation

Use the correct interface by fixing the mentioned issues.

Issue S-19: Spec: Wrong `L2ToL1MessagePasser` interface

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/92

Found by

cmichel, lemonmon

Summary

Wrong `L2ToL1MessagePasser` interface in the specs.

Vulnerability Detail

The withdrawals.md specification file shows a wrong `L2ToL1MessagePasser` interface:

- `function nonce() view external returns (uint256);` does not exist. It should be `function messageNonce() view external returns (uint256);`

```
interface L2ToL1MessagePasser {
    event MessagePassed(
        uint256 indexed nonce, // this is a global nonce value for all
↪   withdrawal messages
        address indexed sender,
        address indexed target,
        uint256 value,
        uint256 gasLimit,
        bytes data,
        bytes32 withdrawalHash
    );

    event WithdrawerBalanceBurnt(uint256 indexed amount);

    function burn() external;

    function initiateWithdrawal(address _target, uint256 _gasLimit, bytes memory
↪   _data) payable external;
   // @audit it's called messageNonce
    function nonce() view external returns (uint256);

    function sentMessages(bytes32) view external returns (bool);
}
```


SHERLOCK

Impact

Users usually go to the docs & specification to see how to integrate a project. Integrating Optimisim's L2ToL1MessagePasser based on the specification will lead to errors.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/3f4b3c328153a8aa03611158b6984d624b17c1d9/specs/withdrawals.md?plain=1#L106

Tool used

Manual Review

Recommendation

Use the correct interface by fixing the mentioned issues.

Issue S-20: Spec: Wrong `L2OutputOracle` interface

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/91

Found by

cmichel, lemonmon

Summary

Wrong `L2OutputOracle` interface in the specs.

Vulnerability Detail

The proposals.md specification file shows a wrong `L2OutputOracle` interface:

- `function getNextBlockNumber() public view returns (uint256)` does not exist. It should be `function nextBlockNumber() public view returns (uint256)`

```
/**
 * @notice The number of the first L2 block recorded in this contract.
 */
uint256 public startingBlockNumber;

/**
 * @notice The timestamp of the first L2 block recorded in this contract.
 */
uint256 public startingTimestamp;

/**
 * @notice Accepts an L2 outputRoot and the timestamp of the corresponding L2
↪  block. The
 * timestamp must be equal to the current value returned by `nextTimestamp()` in
↪  order to be
 * accepted.
 * This function may only be called by the Proposer.
 *
 * @param _l2Output      The L2 output of the checkpoint block.
 * @param _l2BlockNumber The L2 block number that resulted in _l2Output.
 * @param _l1Blockhash   A block hash which must be included in the current
↪  chain.
 * @param _l1BlockNumber The block number with the specified block hash.
*/
  function proposeL2Output(
      bytes32 _l2Output,
      uint256 _l2BlockNumber,
```


SHERLOCK

```
        bytes32 _l1Blockhash,
        uint256 _l1BlockNumber
    )

/**
 * @notice Deletes all output proposals after and including the proposal that
 ↪  corresponds to
 *         the given output index. Only the challenger address can delete
 ↪  outputs.
 *
 * @param _l2OutputIndex Index of the first L2 output to be deleted. All outputs
 ↪  after this
 *                       output will also be deleted.
 */
function deleteL2Outputs(uint256 _l2OutputIndex) external

/**
 * @notice Computes the block number of the next L2 block that needs to be
 ↪  checkpointed.
 */
 // @audit is called `function nextBlockNumber() public view returns (uint256)`
function getNextBlockNumber() public view returns (uint256)
```

Impact

Users usually go to the docs & specification to see how to integrate a project. Integrating Optimisim's L2OutputOracle based on the specification will lead to errors.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/3f4b3c328153a8aa03611158b6984d624b17c1d9/specs/proposals.md?plain=1#L142

Tool used

Manual Review

Recommendation

Use the correct interface by fixing the mentioned issues.

Issue S-21: Spec: Wrong `CrossDomainMessenger` interface

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/90

Found by

Bahurum, cmichel, lemonmon

Summary

Wrong `CrossDomainMessenger` interface in the specs.

Vulnerability Details

The messengers.md specification file shows a wrong `CrossDomainMessenger` interface:

- `function otherMessenger() view external returns (address);` does not exist. It should be `function OTHER_MESSENGER() view external returns (address);`

```
interface CrossDomainMessenger {
    event FailedRelayedMessage(bytes32 indexed msgHash);
    event RelayedMessage(bytes32 indexed msgHash);
    event SentMessage(address indexed target, address sender, bytes message,
↪   uint256 messageNonce, uint256 gasLimit);

    function MESSAGE_VERSION() view external returns (uint16);
    function messageNonce() view external returns (uint256);
    // @audit is OTHER_MESSENGER
    function otherMessenger() view external returns (address);
    function failedMessages(bytes32) view external returns (bool);
    function relayMessage(uint256 _nonce, address _sender, address _target,
↪   uint256 _value, uint256 _minGasLimit, bytes memory _message) payable
↪   external;
    function sendMessage(address _target, bytes memory _message, uint32
↪   _minGasLimit) payable external;
    function successfulMessages(bytes32) view external returns (bool);
    function xDomainMessageSender() view external returns (address);
}
```

Impact

Users usually go to the docs & specification to see how to integrate a project. Integrating Optimisim's CrossDomainMessenger based on the specification will lead to errors.

SHERLOCK

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/3f4b3c328153a8aa03611158b6984d624b17c1d9/specs/messengers.md?plain=1#L38

Tool used

Manual Review

Recommendation

Use the correct interface by fixing the mentioned issues.

Issue S-22: Spec: Wrong Deposited Transaction Type encoding

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/89

Found by

cmichel, tnch, shw

Summary

Wrong Deposited Transaction Type encoding in the specs.

Vulnerability Details

The deposits.md specification file says that the deposited transaction type is encoded with "the following fields (rlp encoded in the order they appear here)":

However, they are missing the `isSystemTransaction` bool from the encoding and the order is also different (`data` after `gasLimit`), see:

```
/**
 * @notice RLP encodes the L2 transaction that would be generated when a given
 ↪  deposit is sent
 *         to the L2 system. Useful for searching for a deposit in the L2
 ↪  system. The
 *         transaction is prefixed with 0x7e to identify its EIP-2718 type.
 *
 * @param _tx User deposit transaction to encode.
 *
 * @return RLP encoded L2 deposit transaction.
 */
function encodeDepositTransaction(Types.UserDepositTransaction memory _tx)
    internal
    pure
    returns (bytes memory)
{
    bytes32 source = Hashing.hashDepositSource(_tx.l1BlockHash, _tx.logIndex);
    bytes[] memory raw = new bytes[](8);
    raw[0] = RLPWriter.writeBytes(abi.encodePacked(source));
    raw[1] = RLPWriter.writeAddress(_tx.from);
    raw[2] = _tx.isCreation ? RLPWriter.writeBytes("") :
 ↪  RLPWriter.writeAddress(_tx.to);
    raw[3] = RLPWriter.writeUint(_tx.mint);
    raw[4] = RLPWriter.writeUint(_tx.value);
    raw[5] = RLPWriter.writeUint(uint256(_tx.gasLimit));
```


SHERLOCK

```
    raw[6] = RLPWriter.writeBool(false);
    raw[7] = RLPWriter.writeBytes(_tx.data);
    return abi.encodePacked(uint8(0x7e), RLPWriter.writeList(raw));
}
```

Impact

Users go to the specification to see how to integrate a project. Integrating according to this spec will be wrong.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/3f4b3c328153a8aa03611158b6984d624b17c1d9/specs/deposits.md?plain=1#L55-L68 https://github.com/ethereum-optimism/optimism/blob/3f4b3c328153a8aa03611158b6984d624b17c1d9/packages/contracts-bedrock/contracts/libraries/Encoding.sol#L22

Tool used

Manual Review

Recommendation

Use the correct encoding from the `encodeDepositTransaction` function above.

Issue S-23: Spec: Wrong `StandardBridge` interface

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/88

Found by

cmichel, lemonmon

Summary

Wrong `StandardBridge` interface in the specs.

Vulnerability Details

The bridges.md specification file shows a wrong `StandardBridge` interface:

- `ERC20BridgeFinalized` event is defined twice, this is an invalid interface as compilation will fail with "DeclarationError: Event with same name and parameter types defined twice."
- The function parameter `_extraData` is defined as `bytes memory _extraData` but the `StandardBridge` uses call-data. The encodings are incompatible.

```
interface StandardBridge {
    // @audit ERC20BridgeFinalized event defined twice, this is an invalid
↪   interface as compilation will fail with "DeclarationError: Event with same
↪   name and parameter types defined twice."
    event ERC20BridgeFinalized(address indexed localToken, address indexed
↪   remoteToken, address indexed from, address to, uint256 amount, bytes
↪   extraData);
    event ERC20BridgeFinalized(address indexed localToken, address indexed
↪   remoteToken, address indexed from, address to, uint256 amount, bytes
↪   extraData);
    event ERC20BridgeInitiated(address indexed localToken, address indexed
↪   remoteToken, address indexed from, address to, uint256 amount, bytes
↪   extraData);
    event ETHBridgeFinalized(address indexed from, address indexed to, uint256
↪   amount, bytes extraData);
    event ETHBridgeInitiated(address indexed from, address indexed to, uint256
↪   amount, bytes extraData);
    // @audit everywhere it is `bytes calldata _extraData`, like 5 times
    function bridgeERC20(address _localToken, address _remoteToken, uint256
↪   _amount, uint32 _minGasLimit, bytes memory _extraData) external;
    function bridgeERC20To(address _localToken, address _remoteToken, address
↪   _to, uint256 _amount, uint32 _minGasLimit, bytes memory _extraData) external;
    function bridgeETH(uint32 _minGasLimit, bytes memory _extraData) payable
↪   external;
```


SHERLOCK

```
    function bridgeETHTo(address _to, uint32 _minGasLimit, bytes memory
↪   _extraData) payable external;
    function deposits(address, address) view external returns (uint256);
    function finalizeBridgeERC20(address _localToken, address _remoteToken,
↪   address _from, address _to, uint256 _amount, bytes memory _extraData)
↪   external;
    function finalizeBridgeETH(address _from, address _to, uint256 _amount,
↪   bytes memory _extraData) payable external;
    function messenger() view external returns (address);
    function otherBridge() view external returns (address);
}
```

Impact

Users usually go to the docs & specification to see how to integrate a project. Integrating Optimisim's bridge based on the specification will lead to errors.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/3f4b3c328153a8aa03611158b6984d624b17c1d9/specs/bridges.md?plain=1#L24

Tool used

Manual Review

Recommendation

Use the correct interface by fixing the mentioned issues.

SHERLOCK

Issue S-24: Incorrectness in computing block signing hash allows cross-chain replay attacks

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/67

Found by

Koolex, usmannk

Summary

Incorrectness in computing block signing hash allows cross-chain replay attacks

Vulnerability Detail

The sequencer signs over a message: keccak256(domain ++ chain_id ++ payload_hash). The chain_id is included to prevent replaying the same message over another chain. However, `SigningHash` function fails to ensure the chain_id is included in the message.

```
func SigningHash(domain [32]byte, chainID *big.Int, payloadBytes []byte)
↪  (common.Hash, error) {
    var msgInput [32 + 32 + 32]byte
    // domain: first 32 bytes
    copy(msgInput[:32], domain[:])
    // chain_id: second 32 bytes
    if chainID.BitLen() > 256 {
        return common.Hash{}, errors.New("chain_id is too large")
    }
    chainID.FillBytes(msgInput[32:64])
    // payload_hash: third 32 bytes, hash of encoded payload
    copy(msgInput[32:], crypto.Keccak256(payloadBytes))

    return crypto.Keccak256Hash(msgInput[:]), nil
}
```

If you look at this line:

```
copy(msgInput[32:], crypto.Keccak256(payloadBytes))
```

https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/op-node/p2p/signer.go#L36

It is supposed to copy the encoded payload hash to the *third 32 bytes* of **msgInput**. However, it start from *32nd byte* which would overwrite the *second 32 bytes* allocated for the chain_id.

SHERLOCK

Impact

Any block signed by the sequencer in any chain is valid for other chains. For example, a malicious verifier can pick a message signed for a test chain and gossip it out for P2P on main chain.

Code Snippet

Please create a file `gossip_crosschain_test.go` under **op-node/p2p** directory and add the following code:

```
package p2p

import (
    // "encoding/binary"
    l "log"
    "testing"
    "math/big"

    "github.com/stretchr/testify/require"
    "github.com/ethereum-optimism/optimism/op-node/rollup"
)

func TestSigningHash(t *testing.T) {

    cfg1 := &rollup.Config{
        L2ChainID:         big.NewInt(100),
    }
    cfg2 := &rollup.Config{
        L2ChainID:         big.NewInt(101),
    }

    payloadBytes := []byte("arbitraryData")
    hash, err := SigningHash(SigningDomainBlocksV1,cfg1.L2ChainID,payloadBytes)
    if err != nil {
        l.Println("Error while hashing", err)
    }
    l.Println("hash : ", hash)

    hash2, err2 := SigningHash(SigningDomainBlocksV1,cfg2.L2ChainID,payloadBytes)
    if err2 != nil {
        l.Println("Error while hashing", err2)
    }
    l.Println("hash2: ", hash2)

    require.NotEqual(t, hash, hash2)
}
```

Run the test as follows:

```
go test -run=TestSigningHashCrossChain ./p2p -v
```

The test will fail as the two hashes match:

```
=== RUN   TestSigningHash
2023/01/28 15:57:41 hash :
↪  0x35af5b09dcb95c4ebac34a932a84a70d6aa97dece830972ce9d0affb7cbaea30
2023/01/28 15:57:41 hash2:
↪  0x35af5b09dcb95c4ebac34a932a84a70d6aa97dece830972ce9d0affb7cbaea30
    gossip_attack_test.go:35:
            Error Trace:    /share/2023-01-optimism-koolexcrypto/optimism/op-nod
            ↪  e/p2p/gossip_attack_test.go:35
            Error:          Should not be: 0x35af5b09dcb95c4ebac34a932a84a70d6aa
            ↪  97dece830972ce9d0affb7cbaea30
            Test:           TestSigningHash
--- FAIL: TestSigningHash (0.00s)
FAIL
FAIL    github.com/ethereum-optimism/optimism/op-node/p2p   0.634s
FAIL
```

Tool used

Manual Review

Recommendation

Just replace this

```
copy(msgInput[32:], crypto.Keccak256(payloadBytes))
```

with

```
copy(msgInput[64:], crypto.Keccak256(payloadBytes))
```

Run the test above again and it should pass successfully.

Discussion

rcstanciu

Comment from Optimism

Reason: The chain ID is indeed being clobbered by the payloadBytes. DDoS risk is low, however, so this is more of a low than a medium.

koolexcrypto

Escalate for 150 USDC

Speaking generally, cross-chain replay was always a critical/high issue. However, in this case here, I believe it should be considered as a medium (at least) since DDoS risk can not be disregarded. Although Optimism's judge acknowledged the DDoS risk, I disagree with the low severity assessment for the following reasons:

1. As you might agree, It is difficult to measure the extent of DDoS attack given the short time of the contest. nevertheless, the issue could cause unpredictable bugs causing serious impacts in future.
2. Since it's planned to decentralise the sequencer ultimately by supporting multiple concurrent sequencers (check refs below), the risk is even higher. Imagine the following (note: this is very simplified just to explain the impact roughly):
    - Assume **DDoS_risk** is the risk level of making the network too busy. **seq(x)** is sequencer x. **count** is how many sequencers we have. **impact(seq(x))** is the potential impact caused by sequencer x. **T(x)** is the count of blocks signed by sequencer x in other chains (e.g. Optimism Goerli). *The higher the T, the higher the impact. If T = 0, there is basically no impact*

      Let $impact(seq(x)) = f(T(x))$,then

      DDoS_risk = $\sum_{n=1}^{count} impact(seq(n))$

As you can notice, once we have multiple sequencers, the risk gets higher.

3. It's also worth mentioning that if it happens that a sequencer produces blocks for Optimism and a forked chain, then one can gossip (in Optimism) those blocks signed for the forked chain. Have a look at this forked chain (VMeta3 chain): https://github.com/VMETA3/vmeta3-chain/blob/master/op-node/p2p/signer.go#L25-L35 It has exactly the same code.
4. Even though the code was audited multiple times, the issue was still overlooked since (at least) May 16, 2022. This shows somewhat the error could possibly stay there till after decentralising the sequencer. Check: https://github.com/ethereum-optimism/optimism/commits/2006a2f5ed921b1069446abbae805a037541dcab/op-node/p2p/signer.go

Moreover, as per Optimism specs:

> The signature is a secp256k1 signature, and signs over a message: keccak256(domain ++ chain_id ++ payload_hash)

Still, the issue was marked as non-reward even though it goes under the "Specification errors" category described in the contest regardless of the reasoning above.

References/Links:

- decentralizing the sequencer
  - https://community.optimism.io/docs/protocol/#decentralizing-the-sequencer
  - https://www.optimism.io/about
- block signatures specs https://github.com/ethereum-optimism/optimism/blob/develop/specs/rollup-node-p2p.md#block-signatures
- History of the code (specifically signer.go) https://github.com/ethereum-optimism/optimism/commits/2006a2f5ed921b1069446abbae805a037541dcab/op-node/p2p/signer.go
- The issue was fixed https://github.com/ethereum-optimism/optimism/pull/4864

Please note that there is a duplicate of the issue and the escalation should be applied to both Issue #125. The issue is also escalated.

sherlock-admin

> Escalate for 150 USDC
>
> Speaking generally, cross-chain replay was always a critical/high issue. However, in this case here, I believe it should be considered as a medium (at least) since DDoS risk can not be disregarded. Although Optimism's judge acknowledged the DDoS risk, I disagree with the low severity assessment for the following reasons:
>
> 1. As you might agree, It is difficult to measure the extent of DDoS attack given the short time of the contest. nevertheless, the issue could cause unpredictable bugs causing serious impacts in future.
> 2. Since it's planned to decentralise the sequencer ultimately by supporting multiple concurrent sequencers (check refs below), the risk is even higher. Imagine the following (note: this is very simplified just to explain the impact roughly):
>
>   - Assume **DDoS_risk** is the risk level of making the network too busy. **seq(x)** is sequencer x. **count** is how many sequencers we have. **impact(seq(x))** is the potential impact caused by sequencer x. **T(x)** is the count of blocks signed by sequencer x in other chains (e.g. Optimism Goerli). *The higher the T, the higher the impact. If T = 0, there is basically no impact*
>
>   Let $impact(seq(x)) = f(T(x))$,then
>
>   **DDoS_risk** = $\sum_{n=1}^{count} impact(seq(n))$

SHERLOCK

As you can notice, once we have multiple sequencers, the risk gets higher.

3. It's also worth mentioning that if it happens that a sequencer produces blocks for Optimism and a forked chain, then one can gossip (in Optimism) those blocks signed for the forked chain. Have a look at this forked chain (VMeta3 chain): https://github.com/VMETA3/vmeta3-chain/blob/master/op-node/p2p/signer.go#L25-L35 It has exactly the same code.
4. Even though the code was audited multiple times, the issue was still overlooked since (at least) May 16, 2022. This shows somewhat the error could possibly stay there till after decentralising the sequencer. Check: https://github.com/ethereum-optimism/optimism/commits/2006a2f5ed921b1069446abbae805a037541dcab/op-node/p2p/signer.go

Moreover, as per Optimism specs:

> The signature is a secp256k1 signature, and signs over a message: keccak256(domain ++ chain_id ++ payload_hash)

Still, the issue was marked as non-reward even though it goes under the "Specification errors" category described in the contest regardless of the reasoning above.

References/Links:

- decentralizing the sequencer
  - https://community.optimism.io/docs/protocol/#decentralizing-the-sequencer
  - https://www.optimism.io/about
- block signatures specs https://github.com/ethereum-optimism/optimism/blob/develop/specs/rollup-node-p2p.md#block-signatures
- History of the code (specifically signer.go) https://github.com/ethereum-optimism/optimism/commits/2006a2f5ed921b1069446abbae805a037541dcab/op-node/p2p/signer.go
- The issue was fixed https://github.com/ethereum-optimism/optimism/pull/4864

Please note that there is a duplicate of the issue and the escalation should be applied to both Issue #125. The issue is also escalated.

You've created a valid escalation for 150 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new**

comment).

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Evert0x

Escalation accepted.

Accepting as a specification issue.

Although it's an issue that needs to be addressed, the likelihood that it will have an impact is so insignificant that it's considered a non-medium severity issue, labeling as spec issue because of section in escalation.

sherlock-admin

> Escalation accepted.
>
> Accepting as a specification issue.
>
> Although it's an issue that needs to be addressed, the likelihood that it will have an impact is so insignificant that it's considered a non-medium severity issue, labeling as spec issue because of section in escalation.

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

Issue S-25: Owner has multiple meanings when proxy is involved

Source: https://github.com/sherlock-audit/2023-01-optimism-judging/issues/27

Found by

keccak123

Summary

The word "owner" is overloaded and refers to more than one address in the proxies used in optimism bedrock.

Vulnerability Detail

The proxy contract used for several contracts, including L1CrossDomainMessenger, confuses the meaning of owner in the code and in the spec. In the proxy contract, there is `OWNER_KEY`, but this storage slot actually stores the admin of the proxy and is retrieved by calling `admin`. This meaning of owner is more confusing because L1CrossDomainMessenger, the implementation contract behind the proxy, inherits OwnableUpgradeable and has an `owner` function, a `transferOwnership` function, and an `onlyOwner` modifier.

The overloading of "owner" makes all documentation with the word "owner" confusing, and sometimes contradictory, in the specifications and code natspec. One specific example of this contradiction is from the predeploy spec

> ProxyAdmin Address: 0x4200000000000000000000000000000000000018
>
> The ProxyAdmin is the owner of all of the proxy contracts set at the predeploys. It is itself behind a proxy. The owner of the ProxyAdmin will have the ability to upgrade any of the other predeploy contracts.

The first time the word "owner" is used, it actually refers to the proxy admin. This is seen by calling `admin` on L2CrossDomainMessenger: `cast call 0x4200000000000000000000000000000000000007 "admin()(address)" --rpc-url https://goerli.optimism.io ->` 0x4200000000000000000000000000000000000018. Using the same admin meaning for owner, like the first time the word is used, returns the ProxyAdmin address, which does not make sense in the context of what the spec is trying to explain: `cast call 0x4200000000000000000000000000000000000018 "admin()(address)" --rpc-url https://goerli.optimism.io ->` 0x4200000000000000000000000000000000000018. But the second time the word "owner" is used, it refers to the owner behind the proxy, in the implementation

SHERLOCK

contract, which is seen in this cast call: `cast call 0x4200000000000000000000000000000000000018 "owner()(address)" --rpc-url https://goerli.optimism.io ->` 0xf80267194936da1E98dB10bcE06F3147D580a62e. This explanation of the spec should replace the first time the word "owner" is used with the word "admin".

Impact

The meaning of owner is ambiguous in many contracts because there owner can refer to the return value of `admin` or the return value of `owner`.

Code Snippet

The natspec for `changeAdmin` and `admin` in the proxy shows how the word owner is used to refer to the proxy admin, confusing the two terms https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/Proxy.sol#L111-L126

The natspec and variable name in L1CrossDomainMessenger uses owner to refer to a different value https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/L1CrossDomainMessenger.sol#L38

The predeploy spec for ProxyAdmin summarizes this confusion by using the word "owner" two times to refer to different values https://github.com/sherlock-audit/2023-01-optimism/blob/main/optimism/specs/predeploys.md#proxyadmin

Tool used

Manual Review

Recommendation

Disambiguate the terms `admin` and `owner`. The proxyadmin spec explanation should replace the first time the word "owner" is used with the word "admin". Consider renaming `OWNER_KEY` in the proxy contract to `ADMIN_KEY` and remove the word owner from the proxy contract.